The Fastest 1.3.6 User's Guide

Automating Software Testing

Maximiliano Cristiá
mcristia@flowgate.net

Pablo Rodríguez Monetti
prodriguez@flowgate.net

Pablo Albertengo
palbertengo@flowgate.net

Flowgate Consulting
Maipú 778 Piso 1 Oficina 10
(2000) Rosario
Argentina

Release date: November 2010

Contents

1 What's New on This Version **4**

2 Introduction to Model-Based Testing **5**

3 The Test Template Framework **6**
3.1 TTF Key Concepts . . . 7
3.1.1 Input Space . . . 7
3.1.2 Valid Input Space . . . 7
3.1.3 Test Class . . . 7
3.1.4 Testing Tactic . . . 7
3.1.5 Testing Tree . . . 8
3.1.6 Pruning Testing Trees . . . 8
3.1.7 Abstract Test Case . . . 9

4 Fastest Architecture in a Nutshell **10**

5 An Example **11**
5.1 The Model – A Z Specification . . . 11
5.2 Running Fastest and Loading the Specification . . . 11
5.3 Generating the Testing Tree . . . 11
5.4 Pruning the Testing Tree . . . 13
5.5 Finding Abstract Test Cases . . . 14

6 Tips on Writing Z Models for Fastest **17**
6.1 The Z Notation Is Not Fully Supported . . . 17
6.2 Fastest Conforms to the Z ISO Standard . . . 17
6.3 Fastest Is Meant to be Used for Unit Testing . . . 17
6.4 Be Careful in Testing Compound Operations . . . 18
6.5 Do Not Include the State Invariant in the State Schema . . . 19
6.6 Keep the State Schema Focused on a Set of Related Operations . . . 19
6.7 Avoid Using Quantifiers . . . 20
6.8 Replace Schema Types with Functions . . . 20
6.9 Avoid Axiomatic Definitions . . . 21
6.10 Avoid Arbitrary Numeric Constants . . . 21

7 User's Manual **23**
7.1 Installing and Executing Fastest . . . 23
7.1.1 Running Fastest and Entering Commands . . . 23
7.2 Steps of a Testing Campaign . . . 24
7.3 Installing and Declaring Testing Servers . . . 25
7.4 Loading a Specification and Selecting Schemas . . . 26
7.5 Applying Testing Tactics and Generating Testing Trees . . . 26
7.5.1 Disjunctive Normal Form . . . 27
7.5.2 Standard Partition (Fastest's name SP) . . . 27
7.5.3 Free Type (Fastest's name FT) . . . 29
7.5.4 In Set Extension (Fastest's name ISE) . . . 29
7.5.5 Proper Subset of Set Extension (Fastest's name PSSE) . . . 29
7.5.6 Subset of Set Extension (Fastest's name SSE) . . . 29
7.5.7 Numeric Ranges (Fastest's name NR) . . . 30

7.5.8 Mandatory Test Set (Fastest's name MTS) . . . 30
7.5.9 Weak Existential Quantifier (Fastest's name WEQ) . . . 30
7.5.10 Strong Existential Quantifier (Fastest's name SEQ) . . . 32
7.6 Dealing with Axiomatic Definitions . . . 32
7.6.1 Basic Types . . . 33
7.6.2 Symbolic Constants . . . 33
7.6.3 Equalities . . . 33
7.6.4 All Other Declarations . . . 34
7.6.5 Command `setaxdef` . . . 34
7.7 Pruning Testing Trees . . . 35
7.7.1 Automatic Pruning . . . 35
7.7.2 Elimination Theorems . . . 36
7.7.3 Loading the Elimination Theorem Library . . . 40
7.7.4 Semiautomatic Pruning . . . 40
7.7.5 Manual Pruning . . . 40
7.8 Generating Abstract Test Cases . . . 41
7.9 Controlling the Generation of Abstract Test Cases . . . 41
7.10 Defining Finite Models . . . 42
7.10.1 Default Finite Models . . . 43
7.10.2 Command `setfinitemodel` . . . 43
7.11 Exploring and Saving the Results . . . 45
7.12 How to Quit Fastest . . . 47

A Z Features Unsupported by Fastest **48**

B Test Classes Generated for *KeepMaxReading* **49**

C Standard Partitions **51**
C.1 Sets . . . 51
C.2 Integers . . . 52
C.3 Relations . . . 53

D Elimination Theorems **55**

1 What's New on This Version

- Command `loadelimtheorems` has been added.
- In elimination theorems, set extensions can be better expressed through the directive `\se`.
- The general pruning algorithm was slightly changed (see section 7.7.1).
- The elimination theorem library delivered with this version is included in Appendix D.
- This version partially supports axiomatic definitions (see section 7.6).
- Commands `setaxdef`, `showaxdef` and `showaxdefvalues` have been added.
- Limited support for four new testing tactics have been added: Numeric Ranges (NR), Mandatory Test Set (MTS), Weak Existential Quantifier (WEQ) and Strong Existential Quantifier (SEQ) (see sections 7.5.7, 7.5.8, 7.5.9 and 7.5.10, respectively).
- Testing tactics ISE, PSSE and SSE have been fixed to support expressions at the left of $\in$, $\subset$ or $\subseteq$, and not only variables.
- Section 6 was added to document some tips on how to write Z models more suitable to be loaded on Fastest.
- Appendix A lists some Z features *still* unsupported by Fastest.
- A configuration variable, `MAX_EVAL`, has been added to limit the number of evaluations when searching for abstract test cases (see section 7.8).
- A configuration variable, `DEF_SIZE_FINITE_SET`, has been added to set the size of finite sets which latter are used to find abstract test cases (see section 7.8).
- It is possible to load specifications where terms are used before their declarations.
- Command-line editing features (like tab completion) have been added (see section 7.1.1).
- In this version, it is possible to apply a testing tactic to a selected sub-tree of a testing tree (and not just to the entire tree as in previous versions).
- The performance of the abstract test case generation algorithm was improved.

2 Introduction to Model-Based Testing

- Testing could be the most costly phase of a software development project.
- We can use formal methods to make testing almost automatic, thus changing the cost of testing by the cost of formalization—which is reported to be quite less expensive.
- Model-based testing (MBT) uses a formal specification to generate test cases and to verify whether they found errors in the program or not.
- Figure 1 depicts one of the possible MBT methodologies, a part of which is implemented by Fastest. The methodology is based on [SC96], [HP95] and [Sto93].
- Testing starts by writing a model of the software under test.
- At the beginning, the model is used to generate test cases.
- At the end, the model is used as an oracle.
- The process is very automatic if we have the model.
- The current version of Fastest only implements the *generation* phase of Figure **??**.
- Fastest is more suitable for unit testing.
- **The MBT methodology implemented by Fastest, called Test Template Framework (see next section), uses Z formal specifications.**

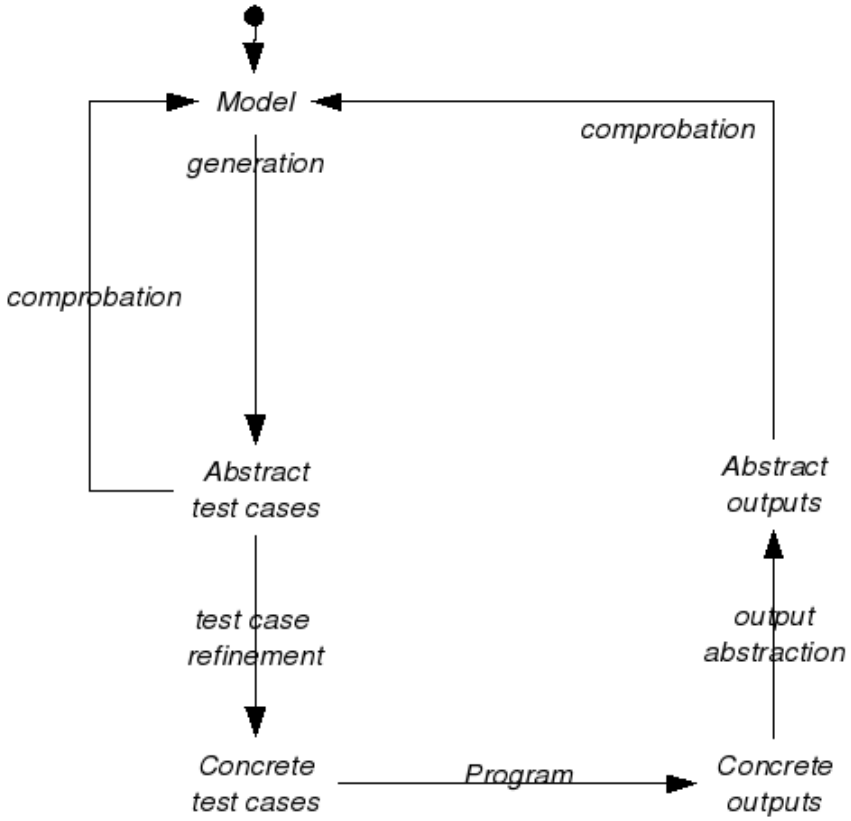


Figure 1: Model-based testing methodology: general view.

3 The Test Template Framework

Fastest implements the Test Template Framework (TTF). TTF is a MBT framework proposed by Phil Stocks and David Carrington in [SC96] and [Sto93]. Although the TTF was meant to be notation-independent, the original presentation was made using the Z formal notation. It is one of the few MBT frameworks approaching unit testing.

The TTF deals with the *generation* phase shown in Figure 1. In this framework, each operation within the specification is analysed to derive or generate *abstract test cases*. This analysis consists of the following steps, roughly depicted in Figure 2:

1. Define the *input space* (IS) of each operation.
2. Derive the *valid input space* (VIS) from the IS of each operation.
3. Apply one or more *testing tactics*[1], starting from each VIS, to build a *testing tree* for each operation. Testing trees are populated with nodes called *test classes*.
4. *Prune* each of the resulting testing trees.
5. Find one or more *abstract test cases* from each leaf in each testing tree.

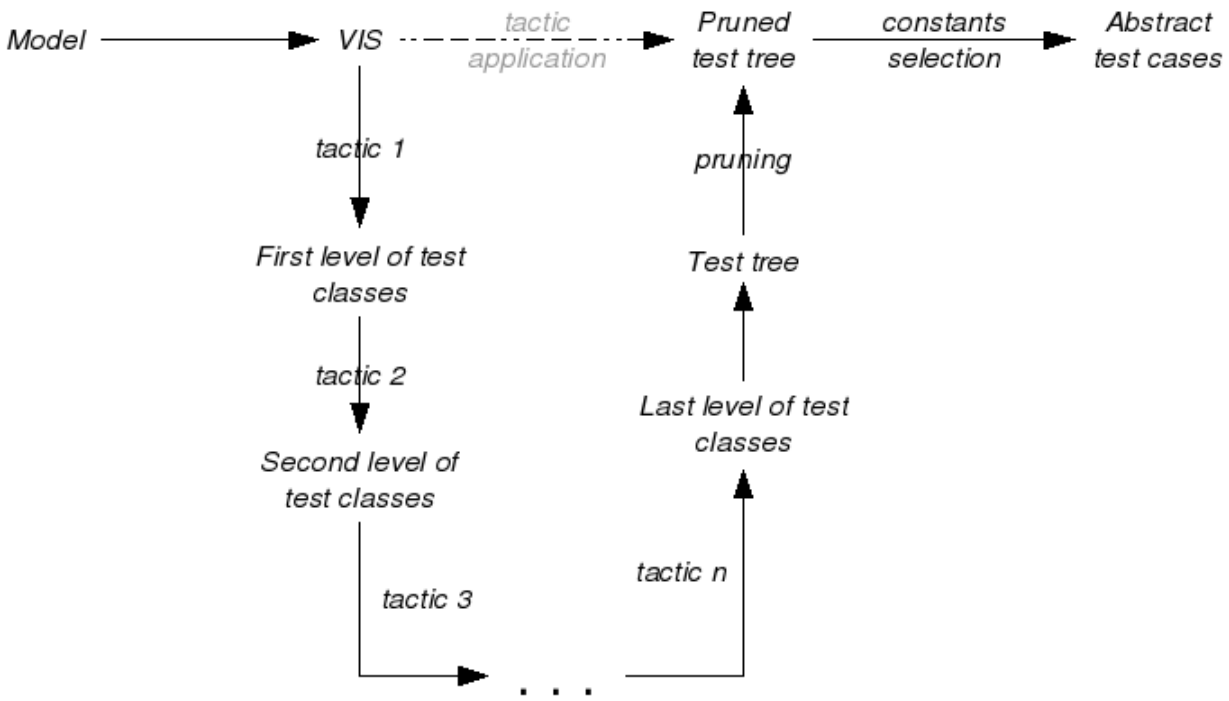


Figure 2: The Test Template Framework process is a detailed view of the *generation* phase shown in Figure 1.

One of the main advantages of the TTF is that all of these concepts are expressed in the same notation of the specification, i.e. the Z notation. Hence, the engineer has to know only one notation to perform the analysis down to the generation of abstract test cases.

The concepts introduced above are explained in the next section. How Fastest implements the TTF, is explained by means of an example in Section 5.

[1]Stocks and Carrington use the term "testing strategies" in [SC96].

1. $S = \varnothing,\ T = \varnothing$	5. $S \neq \varnothing,\ T \neq \varnothing,\ S \subset T$
2. $S = \varnothing,\ T \neq \varnothing$	6. $S \neq \varnothing,\ T \neq \varnothing,\ T \subset S$
3. $S \neq \varnothing,\ T = \varnothing$	7. $S \neq \varnothing,\ T \neq \varnothing,\ T = S$
4. $S \neq \varnothing,\ T \neq \varnothing,\ S \cap T = \varnothing$	8. $S \neq \varnothing,\ T \neq \varnothing,\ S \cap T \neq \varnothing,\ \neg\,(S \subseteq T),\ \neg\,(T \subseteq S),\ S \neq T$

Figure 3: Standard partition for $S \cup T$, $S \cap T$ y $S \setminus T$.

3.1 TTF Key Concepts

In this section the main concepts defined by the TTF are described.

3.1.1 Input Space

Let Op be a Z operation. Let $x_1 \ldots x_n$ be all the input and (non-primed) state variables declared in Op (or in its included schemas), and $T_1 \ldots T_n$ their corresponding types. The *Input Space* (IS) of Op, written IS_{Op}, is the Z schema box defined by $[x_1 : T_1 \ldots x_n : T_n]$.

3.1.2 Valid Input Space

Let Op be a Z operation. Let pre Op be the precondition of Op. The *Valid Input Space* (VIS) of Op, written VIS_{Op}, is the Z schema box defined by $[IS_{Op} \mid \text{pre } Op]$.

3.1.3 Test Class

Informally, test classes are sets of abstract test cases defined by comprehension; hence each test class is identified by a predicate. Test classes are also called test objectives [UL06], test templates [SC96], test targets and test specifications.

Let Op be a Z operation and let P be any predicate depending on one or more of the variables defined in VIS_{Op}. Then, the Z schema box $[VIS_{Op} \mid P]$ is a *test class* of Op. Note that this schema is equivalent to $[IS_{Op} \mid \text{pre } Op \wedge P]$. This observation can be generalized by saying that if C_{Op} is a test class of Op, then the Z schema box defined by $[C_{Op} \mid P]$ is also a test class of Op. According to this definition the VIS is also a test class.

If C_{Op} is a test class of Op, then the predicate P in $C'_{Op} == [C_{Op} \mid P]$ is said to be the *characteristic* predicate of C'_{Op} or C'_{Op} is *characterized* by P.

3.1.4 Testing Tactic

In the context of the TTF a *testing tactic* is a means of partitioning any test class of any operation. However, some of the testing tactics used in practice actually do not always generate a partition, in the mathematical sense, of some test classes.

For instance, two testing tactics originally proposed for the TTF are the following:

- Disjunctive Normal Form (DNF). By applying this tactic the operation is written in Disjunctive Normal Form and the test class is divided in as many test classes as terms are in the resulting operation's predicate. The predicate added to each new test class is the precondition of one of the terms in the operation's predicate.

- Standard Partitions (SP). This tactic uses a predefined partition of some mathematical operator [Sto93]. For example, the partition shown in Figure 3 is a good partition for expressions of the form $S \spadesuit T$ where $\spadesuit$ is one of $\cup$, $\cap$ and $\setminus$.

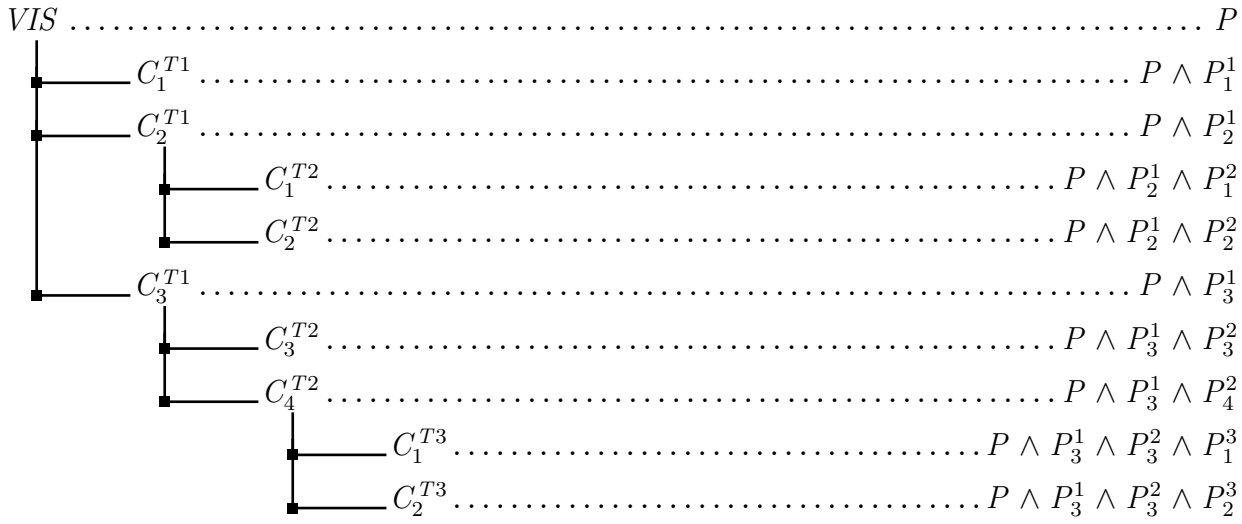

VIS . . . . . P
C_1^{T1} . . . . . $P \wedge P_1^1$
C_2^{T1} . . . . . $P \wedge P_2^1$
C_1^{T2} . . . . . $P \wedge P_2^1 \wedge P_1^2$
C_2^{T2} . . . . . $P \wedge P_2^1 \wedge P_2^2$
C_3^{T1} . . . . . $P \wedge P_3^1$
C_3^{T2} . . . . . $P \wedge P_3^1 \wedge P_3^2$
C_4^{T2} . . . . . $P \wedge P_3^1 \wedge P_4^2$
C_1^{T3} . . . . . $P \wedge P_3^1 \wedge P_3^2 \wedge P_1^3$
C_2^{T3} . . . . . $P \wedge P_3^1 \wedge P_3^2 \wedge P_2^3$

Figure 4: The predicate of a test class at some level is the conjunction of the predicate of its parent test class and its own predicate.

As can be noticed, standard partitions might be changed according to how much testing the engineer wants to perform.

3.1.5 Testing Tree

The application of a testing tactic to the VIS generates some test classes. If some of these test classes are further partitioned by applying one or more testing tactics, a new set of test classes is obtained. This process can continue by applying testing tactics to the test classes generated so far. Evidently, the result of this process can be drawn as a tree with the VIS as the root node, the test classes generated by the first testing tactic as its children, and so on. In other words, test classes' predicates obey the relationship depicted in Figure 4. A consequence of this relationship is that the deeper the tree, the more accurate and discovering the test cases. As was noted by Stocks and Carrington in [SC96], the Z notation can be used to build the tree, as follows.

$$
\begin{aligned}
&VIS == [IS \mid P] \\
&C_1^{T_1} == [VIS \mid P_1^1] \\
&C_2^{T_1} == [VIS \mid P_2^1] \\
&C_1^{T_2} == [C_2^{T_1} \mid P_1^2] \\
&C_2^{T_2} == [C_1^{T_1} \mid P_2^2] \\
&C_3^{T_1} == [VIS \mid P_3^1] \\
&C_3^{T_2} == [C_3^{T_1} \mid P_3^2] \\
&C_4^{T_2} == [C_3^{T_1} \mid P_4^2] \\
&C_1^{T_3} == [C_4^{T_2} \mid P_1^3] \\
&C_2^{T_3} == [C_4^{T_2} \mid P_2^3]
\end{aligned}
$$

3.1.6 Pruning Testing Trees

In general a test class' predicate is a conjunction of two or more predicates. It is likely, then, that some test classes are empty because their predicates are contradictions. These test classes must be pruned from the testing tree because they represent impossible combinations of input values, i.e. no

abstract test case can be derived out of them. In this way, pruning the initial testing tree saves CPU time because it avoids searching abstract test cases in empty sets.

3.1.7 Abstract Test Case

An abstract test case is an element belonging to a test class. The TTF prescribes that abstract test cases should be derived only from the leaves of the testing tree. Abstract test cases can also be written as Z schema boxes. Let Op be some operation, let VIS_{Op} be the VIS of Op, let $x_1 : T_1 \ldots x_n : T_n$ be all the variables declared in VIS_{Op}, let C_{Op} be a (leaf) test class of the testing tree associated to Op, let $P_1 \ldots P_m$ be the characteristic predicates of each test class from C_{Op} up to VIS_{Op} (by following the edges from child to parent), and let $v_1 : T_1 \ldots v_n : T_n$ be n constant values satisfying $P_1 \wedge \ldots \wedge P_m$. Then, an abstract test case of C_{Op} is the Z schema box defined by $[C_{Op} \mid x_1 = v_1 \wedge \ldots \wedge x_n = v_n]$.

4 Fastest Architecture in a Nutshell

- Fastest architecture was guided by:
  - Performance, because calculating thousands of test cases can be very time consuming.
  - Modificability, because we don't know yet what features industry could need.
  - Documentability, because to test must mean to document.
- Hence, we combined two different architectural styles and we used open formats and tools in our interfaces:
  - Client/Server, so we distribute test case calculation.
  - Implicit Invocation, so we can add, modify and remove components as new and more sophisticated requirements arise.
  - Latex is used to read specifications and to generate testing trees, test classes, abstract test cases, etc.
  - Fastest has been integrated with the Community Z Tools project (CZT), to avoid duplicate efforts in programming core modules.
- Fastest' process structure is shown in Figure 5.
  - Clients interact with the user and generate and prune testing trees.
  - All of the other functions are performed on the servers.
  - The knowledge base server stores testing tactics, abstract test cases, refinement functions, etc. so testers can use them for re-testing within a given project and for testing in different projects.
    Currently this server is not fully implemented.
- This process structure takes advantage of a quite obvious fact: abstract test cases can be easily and efficiently calculated by a highly parallel system.

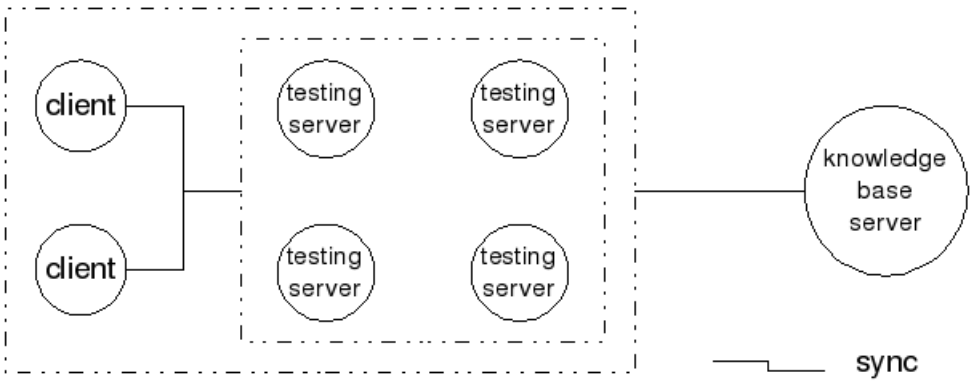


Figure 5: Fastest's process structure is composed by a number of clients and servers, and just one instance of a knowledge-base server.

5 An Example

- We have said that MBT takes as input a formal model of the system to be tested.
- Fastest uses Z specifications.

  The current version does not support the full language, see Appendix A for a list of the unsupported Z features.

5.1 The Model – A Z Specification

- Z is a formal notation useful to specify systems that will use complex data structures and will apply complex transformations over them.
- The Z notation is a textual language based on typed first order logic and discrete mathematics. For standard presentations of the Z language read any of the excellent available books, such as [PST96], [Jac97], [Dil90] and [WD96, available on-line from here].
- A Z model is a state machine where states are defined by state variables and transitions are predicates over those variables[2].

5.2 Running Fastest and Loading the Specification

- We will apply Fastest to test a function that must keep the highest readings taken from a set of sensors.
- The Z formal specification for the function mentioned above is depicted in Figure 6. We did not write the state invariant in this example in order to simplify the presentation.
- To run Fastest[3]: open a command window, move to the directory where you installed Fastest and execute the following command.

  ```
  java -Xss8M -Xms512m -Xmx512m -jar fastest.jar
  ```

- Assuming the specification is stored in a file called `sensors-simp.tex`, and if this file is stored in the `doc` directory under Fastest's root directory, load the specification with:

  ```
  loadspec doc/sensors-simp.tex
  ```

5.3 Generating the Testing Tree

- Before generating a testing tree you need to select one or more operations to test. In our example we select *KeepMaxReading*.

  ```
  selop KeepMaxReading
  ```

- The testing tree depends on the tactics you apply and the order they are applied.
- In this case we will apply two testing tactics:

[2] We assume the reader is familiar with the Z formal notation.
[3] In this section we just give an overview of how to use Fastest. Section 7 is a thorough user manual.

$[SENSOR]$

$MaxReadings == [smax : SENSOR \pfun \mathbb{Z}]$

KeepMaxReadingOk

$\Delta MaxReadings$
$s? : SENSOR;\ r? : \mathbb{Z}$

$s? \in \mathrm{dom}\ smax$
$smax\ s? < r?$
$smax' = smax \oplus \{s? \mapsto r?\}$

*KeepMaxReadingE*1 ==
$[\Xi MaxReadings;\ s? : SENSOR \mid$
$s? \notin \mathrm{dom}\ smax]$

*KeepMaxReadingE*2

$\Xi MaxReadings$
$s? : SENSOR;\ r? : \mathbb{Z}$

$s? \in \mathrm{dom}\ smax$
$r? \leq smax\ s?$

KeepMaxReading ==
KeepMaxReadingOk
$\lor$ *KeepMaxReadingE*1
$\lor$ *KeepMaxReadingE*2

Figure 6: A Z specification of an operation that keeps the maximum values read from a set of sensors.

- Disjunctive Normal Form (DNF). It's applied by default.
- Standard Partitions (SP). We will apply it to the expression $smax\ s? < r?$ present in schema *KeepMaxReadingOk*.

Then, you only need to issue one command to apply the last tactic.

```
addtactic KeepMaxReading SP < smax~s? < r?
```

- Once you have added all the tactics you want, the testing tree is generated with:

```
genalltt
```

- The resulting testing tree is the one shown in Figure 7. It can be printed with:

```
showtt
```

- Each node in the testing tree (i.e. a test class) is also a Z schema, describing the conditions for selecting a test case. The content of those Z schema can be displayed with the following command.

```
showsch -tcl
```

You can take a look at these test classes in Appendix B.

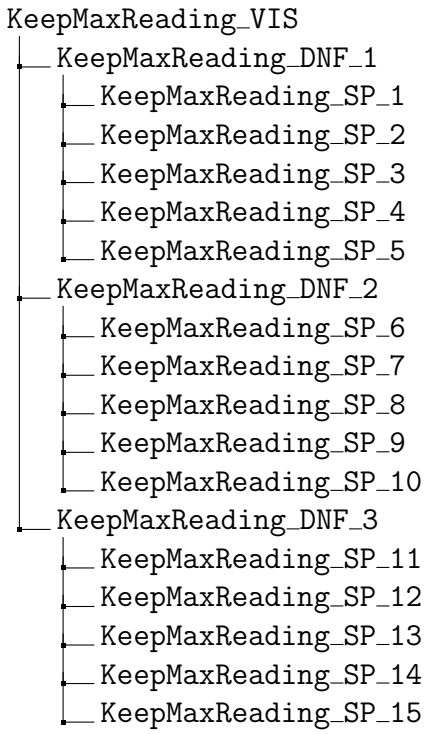


Figure 7: Testing tree for *KeepMaxReading*.

5.4 Pruning the Testing Tree

- It is important to prune from the testing trees all the empty test classes.
- In Fastest pruning can be done manually or automatically. We strongly suggest to use the automatic strategy, and perhaps complement it with the manual one.
- The simplest form, and the one you should try first, to use the automatic pruning is by running the following command.

  `prunett`

  This command can only be run after `genalltt`.
- Fastest will try to prune all the empty test classes, but in some cases it will left some of them hanging from the tree. This is due to the fact that the automatic strategy implements a best-effort algorithm. However, the user can improve this strategy for the current or future projects by adding so called *elimination theorems* to a configuration file –to know more about this go to section 7.7.1.
- After `prunett` is done, you can either explore the remaining test objectives to see if some of them are empty, and then prune them manually, or tell Fastest to find abstract test cases as is explained in the following section –we suggest this last curse of action.
- In the example we are examining, `prunett` actually prunes all the empty test classes, so the user should not prune any node manually.
- The resulting testing tree is shown in Figure 8.
- There are three commands to prune testing trees manually. All of them receive a test class name as parameter.

```
KeepMaxReading_VIS
├─ KeepMaxReading_DNF_1
│   ├─ KeepMaxReading_SP_1
│   ├─ KeepMaxReading_SP_2
│   ├─ KeepMaxReading_SP_3
│   ├─ KeepMaxReading_SP_4
│   └─ KeepMaxReading_SP_5
├─ KeepMaxReading_DNF_2
└─ KeepMaxReading_DNF_3
    ├─ KeepMaxReading_SP_11
    └─ KeepMaxReading_SP_15
```

Figure 8: Testing tree for *KeepMaxReading* after running `prunett`.

- `prunefrom c`: prunes the sub-tree starting at class `c`.
- `prunebelow c`: prunes the sub-tree starting at class `c`, including `c` itself.
- `unprune c`: restores class `c`.

See more about these commands in section 7.7.1.

5.5 Finding Abstract Test Cases

- Now, it is time to try to find one abstract test case in each leaf of the testing tree.
- Fastest accomplishes this by generating a finite model over which it evaluates each leaf's predicate.
  - If some element of the finite model satisfies a test class' predicate, then we have found an abstract test case in that class.
  - If no element in the finite model satisfies the predicate, it can happen because:
    - The predicate is a contradiction (or the test class is empty). This can happen because Fastest is not smart enough to prune all the empty test classes, although it can become smarter if the user teaches it –see section 7.7.1.
    - The finite model is not a appropriate. The predicate is not a contradiction so there are other finite models that can satisfy it.
- By running

  `genalltca`

  Fastest will try to find a test case for each leaf in the testing tree by applying default finite models.
- After some time Fastest finds test cases for almost all the leaves as shown in Figure 9 –this tree is displayed with `showtt` as we have said earlier.
- As you can see, Fastest failed to find abstract test cases for *KeepMaxReading_SP_1* and *KeepMaxReading_SP_9* while it should.

  The reason is that for these test classes Fastest will choose $\{-1, 0, 1\}$ as the finite set for $\mathbb{Z}$. Then, if you take a look at these test classes (run `showsch KeepMaxReading_SP_1`) you can see that both predicates are satisfied either with two strictly positive or negative numbers, but that it is impossible if only $\{-1, 0, 1\}$ is considered.

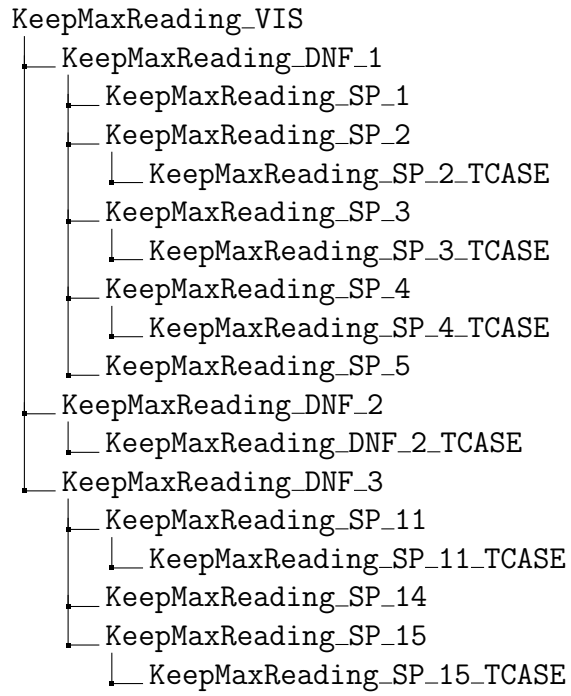


Figure 9: The extended testing tree includes also the schema boxes representing test cases hanging from those leaves for which Fastest was able to find a test case.

KeepMaxReading_SP_1

$$smax : SENSOR \pfun \mathbb{Z}$$
$$s? : SENSOR$$
$$r? : \mathbb{Z}$$

$$s? \in \operatorname{dom} smax$$
$$smax\ s? < r?$$
$$smax\ s? < 0$$
$$r? < 0$$

KeepMaxReading_SP_5

$$smax : SENSOR \pfun \mathbb{Z}$$
$$s? : SENSOR$$
$$r? : \mathbb{Z}$$

$$s? \in \operatorname{dom} smax$$
$$smax\ s? < r?$$
$$smax\ s? > 0$$
$$r? > 0$$

- Fastest gives you the chance to define a finite model for each leaf. You can do it before or after running `genalltca` but always after running `genalltt`. Define finite models for *KeepMaxReading_SP_1* and *KeepMaxReading_SP_5* as follows.

```
setfinitemodel KeepMaxReading_SP_1 -fm "\num == \{\negate 1,\negate 2\}"
setfinitemodel KeepMaxReading_SP_5 -fm "\num == 1 \upto 2"
```

- Now, run `genalltca` again so Fastest can find abstract test cases for those leaves for which there is no test case already.
- By running `showtt` you can see that there are test cases for all leaves.
- To see the abstract test cases run `showsch -tca`. Note that each abstract test case is a Z schema, in which each state and input variable equals a constant value.

KeepMaxReading_SP_1_TCASE

KeepMaxReading_SP_1

$$r? = -1$$
$$smax = \{(sensor0, -2)\}$$
$$s? = sensor0$$

KeepMaxReading_SP_2_TCASE

KeepMaxReading_SP_2

$r? = 0$
$smax = \{(sensor0, -1)\}$
$s? = sensor0$

KeepMaxReading_DNF_2_TCASE

KeepMaxReading_DNF_2

$r? = 0$
$smax = \varnothing$
$s? = sensor0$

KeepMaxReading_SP_3_TCASE

KeepMaxReading_SP_3

$r? = 1$
$smax = \{(sensor0, -1)\}$
$s? = sensor0$

KeepMaxReading_SP_11_TCASE

KeepMaxReading_SP_11

$r? = -1$
$smax = \{(sensor0, -1)\}$
$s? = sensor0$

KeepMaxReading_SP_4_TCASE

KeepMaxReading_SP_4

$r? = 1$
$smax = \{(sensor0, 0)\}$
$s? = sensor0$

KeepMaxReading_SP_15_TCASE

KeepMaxReading_SP_15

$r? = 1$
$smax = \{(sensor0, 1)\}$
$s? = sensor0$

KeepMaxReading_SP_5_TCASE

KeepMaxReading_SP_5

$r? = 2$
$smax = \{(sensor0, 1)\}$
$s? = sensor0$

- This test cases as well as the testing tree and test classes can be saved to disk in LaTeX format with the following commands.
  - `showtt -tcl -o filename`, saves the testing tree.
  - `showtt -o filename`, saves the test classes.
  - `showsch -tca -o filename`, saves the abstract test cases.

6 Tips on Writing Z Models for Fastest

Z is a very general language that can be used for several purposes and specifications can be written in a variety of styles. Although Fastest can work with any kind of Z specifications—provided they are written with the subset of Z currently supported (Appendix A)—it works better if specifications follow some specific rules described in the following sections.

Important!! Working differently as we suggest might lead to performance penalties or even to Fastest being unable to derive abstract test cases. If the specification or some of its operations are very complex, then Fastest could crash when these operations have to be processed. However, if complex operations are treated correctly Fastest might provide very useful information regarding test case design and even abstract test cases.

6.1 The Z Notation Is Not Fully Supported

Before writing a Z specification for Fastest, please, read Appendix A to learn what parts of the Z notation are still unsupported by the tool. The support for these features is being postponed since we consider that they are somewhat superfluous for Fastest's purpose. Hence, you will get a broad idea of what kind of models are best suited for Fastest by first reading the appendix.

6.2 Fastest Conforms to the Z ISO Standard

Fastest parses and type-checks specifications written in the LaTeX mark-up that conforms to the Z ISO Standard [ISO02]. Then, it does not accept Spivey's grammar.

6.3 Fastest Is Meant to be Used for Unit Testing

The main and foremost purpose of Fastest is to be an aid in **unit testing**. Then, specifications should represent units of implementation or they can be decomposed as such. To be more clear, we think in an unit as a subroutine, a function, or a method. Therefore, if your model ends with schema *System* representing the behaviour of the whole system, and you try to "test" *System*, Fastest will probably perform poorly. Likely, *System* is the disjunction of a number of primitive operations each of which, probably, represents a unit of implementation. Hence, you will get the best of Fastest by trying to "test" each of these operations in isolation instead of working directly with *System*.

One important point here is that software design—i.e. decomposing the software into a set of elements, assigning a function to each element and defining their relationships [GJM91]—should guide the specification. In doing so, you will identify a set of modules or components each providing a public interface to the rest of the system. Each module should be simple and small enough to be easily understood; it will probably be implemented by one programmer. The Z specification of such a design should, then, has one state schema per module and one operation schema for each subroutine exported by each module. These are the primitive operations. If you do not have a design or if you do not want to define one before understanding the requirements, then at least, write the specification as a set of primitive operations that are progressively integrated to provide some complex services. In either case, use Fastest to derive test cases for these primitive operations.

Note that full specifications can be given for these primitive operations. In other words, give schemas for both successful and erroneous conditions for each operation, but keep them primitive.

$A_1 == B \land C$

$A_2 == B \fatsemi C$

$A_3 == (B \lor D) \land C$

A_4

$Decl$
$Pred \Rightarrow B'$
$Pred_1 \Rightarrow C'$

$A_5 == [Decl \mid Pred] \land B \land C$

A_6

B
C
$Decl$
$Pred$

Figure 10: Some compound operations. *Decl* is a declaration and *Pred* and $Pred_1$ are a predicates. B, C and D are primitive operations.

6.4 Be Careful in Testing Compound Operations

Although it is a matter of style, specifiers might want to represent that some operation "calls" or "uses" the services of other operations. This is some times achieved by specifying an operation as the conjunction ($\land$) or the composition ($\fatsemi$) of some other operations. In particular, conjunction can be written as schema inclusion. We call these *compound operations*. For instance, consider the compound operations sketched in Figure 10. Let's assume in all cases that B, C and D are primitive operations. Besides, say that `fA1`, `fA2`, `fA3`, `fA4`, `fA5`, `fA6`, `fB`, `fC` and `fD` are the subroutines implementing the corresponding operations. Then, we suggest to work as follows in each case.

Cases A_1 and A_2. These cases are very easy to deal with. Just derive test cases for B and C and not for A_1 and A_2. The point here, as with some of the other cases, is that the correctness of either `fA1` or `fA2` depends solely on the correctness of both `fB` and `fC`. Then, one should derive unit test cases for `fB` and `fC` only and then integrate[4] them to test `fA1` and `fA2`. Since Fastest now is only good for unit testing, then trying to apply it to derive test cases for A_1 and A_2 might not be the best option.

Case A_3. We have distinguished this case from the previous one because we want to emphasize that we think that B and D are two distinct operations and not two schemas of the same operation—like the normal case and an erroneous one. If this is the case, then we think that the best course of action is to work as indicated in the previous paragraph.

Case A_4. B' and C' do not have precondition since all of their variables are primed. Hence, their predicates will not have much influence in abstract test case derivation since test cases are generated from the input space of the operation. For this reason Fastest will not unfold these schema references. In this case, then, the user can work directly with A_4.

Case A_5. We suggest to write A_5 as follows:

$$A_5 == E \land B \land C$$

where E is $[Decl \mid Pred]$, and then to derive test cases for B, C and E and not for A_5. If $[Decl \mid Pred]$ is not named, Fastest will not recognize it as an operation thus making it impossible for the user to work with it.

[4]Integration testing is not implemented yet.

x is intended to be a natural number but $\mathbb{N}$ is not a type, then we need an invariant. An equivalent schema would have been $Naturals == [x : \mathbb{Z}]$ but the invariant would be hidden.

Naturals

$x : \mathbb{Z}$

$0 \leq x$

Decr

$\Delta Naturals$

$x' = x - 1$

No proof is needed to verify that *Decr* preserves the state invariant because state variables are restricted to satisfy it.

(a) Classic style.

Naturals

$x : \mathbb{Z}$

NaturalsInv

Naturals

$0 \leq x$

Decr

$\Delta Naturals$

$x > 0$

$x' = x - 1$

Theorem DecrVerifiesInvariant

$NaturalsInv \wedge Decr \Rightarrow NaturalsInv\,'$

(b) Proof obligation style.

Figure 11: A simple example showing two styles of writing state invariants. Fastest works better with the proof obligation style.

Case A_6. As with the previous case we suggest to rewrite A_6 as follows:

$$E == [Decl_1 \mid Pred]$$
$$A_6 == E \wedge B \wedge C$$

where $Decl_1$ might be different from *Decl* since it may be necessary to add some variables because E does not include B and C, which may add some declarations. If the operation is written in this way, then derive test cases for B, C and E and not for A_6 as we suggested in the previous cases.

6.5 Do Not Include the State Invariant in the State Schema

It is the classic style within the Z community to include the state invariant inside the state schema, as shows the simple example of Figure 11a. However, Fastest works better if the state invariant is not included in the state schema as shown in Figure 11b. This is the style followed by specification languages such as B [Abr96] and TLA+ [Lam02].

Writing the state invariant outside the state schema makes it a proof obligation rather than a state restriction. At the same time, this style avoids implicit preconditions perhaps making the specification clearer to programmers because they do not need to calculate them. But explicit preconditions are the key to input domain partition, which is the fundamental concept behind the TTF. Hence, by writing the state invariant outside the state schema we avoid implicit preconditions, thus, enabling input domain partition.

6.6 Keep the State Schema Focused on a Set of Related Operations

As we have seen, the input space (IS) of a Z operation is defined as the schema declaring all the input and state variables of the operation. An abstract test case is an element belonging to the

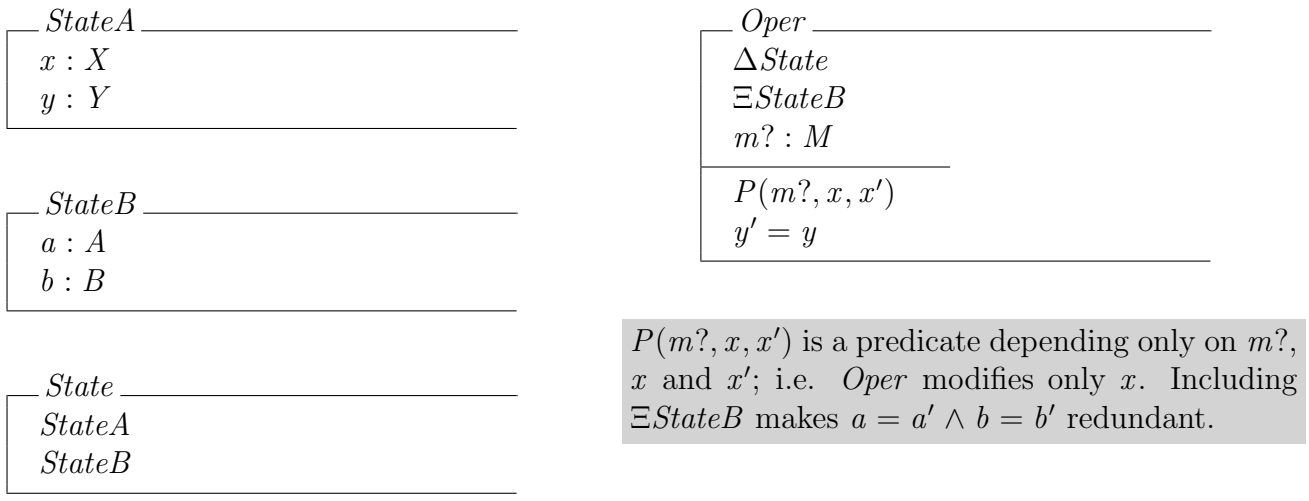


Figure 12: State variables are grouped in state schemas. Operations include Δ of the full state and Ξ of those part of the state that they do not modify.

IS. Then, the more variables in the IS, the longer the abstract test cases. Furthermore, if some of the IS variables are not referenced in the operation's predicate, then it means that these variables are irrelevant for the operation. But they still need to be included in abstract test cases. Hence, it is possible that abstract test cases contain many variables such that only a fraction of them are meaningful to the tester.

It does not matter whether this irrelevant variables appear in predicates such us $var' = var$, there still be an equality like $var = const$ in every abstract test case of the corresponding operation. For instance, it is a common style to divide the state variables in some state schemas that then are joined to define the whole state space of the system, as shown in 12. In this way, specifiers avoid to write many equalities for those variables that the operation does not modify. However, a better strategy if the specification is going to be loaded into Fastest would be to specify *Oper* as follows:

$$Oper == [\Delta StateA;\ m? : M \mid P(m?, x, x') \land y' = y]$$

Clearly, this is possible only because *Oper*'s predicate depends only on the state variables declared in *StateA*. In the extreme case *Oper* can be specified with:

$$Oper == [x, x' : X;\ m? : M \mid P(m?, x, x')]$$

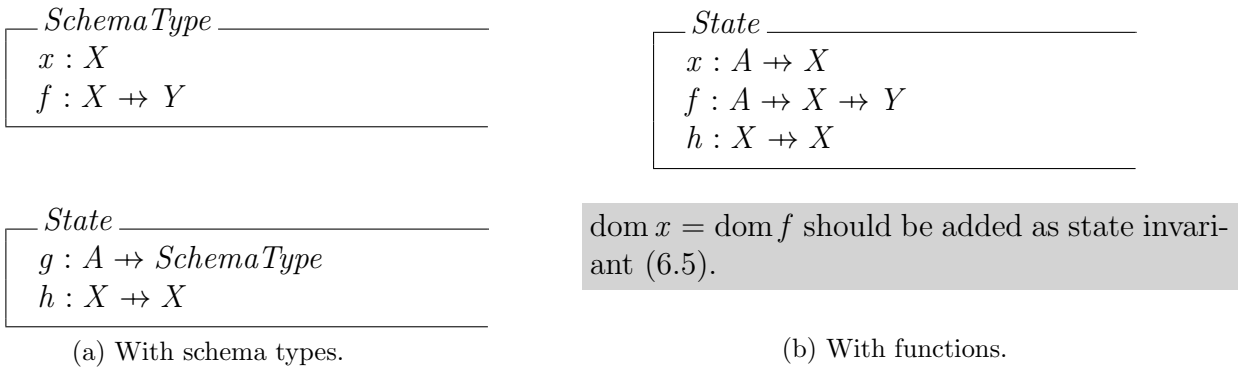
but this implies that abstract test cases derived by Fastest will not mention y. This might be a problem if the unit implementing *Oper* needs some initial value for all of its variables—but perhaps that is an indication of some poor implementation.

6.7 Avoid Using Quantifiers

Quantifiers always complicate software verification. Then, it is a good advice to avoid them as much as possible regardless whether Fastest will be used or not—wisely use the rich mathematical operators provided by Z to avoid many quantifications. Fastest will enter into troubles if it needs to find abstract test cases from test classes whose predicates include quantifiers. However, it will succeed in many cases.

6.8 Replace Schema Types with Functions

Maybe the most important Z feature currently not supported by Fastest are schema types. Schema types are normally used along with operation promotion. Operation promotion talks about compound



(a) With schema types.

(b) With functions.

Figure 13: Replacing schema types with functions. (a) represents the original specification, and (b) represents the alternative without schema types.

operations (6.4), which in turn refers to integration testing—and Fastest is meant to be used for **unit testing** (6.3).

If schema types are indeed needed, there is a chance to replace them with functions as shown in Figure 13.

6.9 Avoid Axiomatic Definitions

Fastest supports axiomatic definitions as described in Section 7.6. However, their presence decreases the level of automation of the tool, so it is better to avoid them as much as possible. Conceptually, axiomatic definitions are parameters of the specification. In other words, the meaning of an specification depends on the particular value assumed for each axiomatic definition. Since test cases are derived from the specification, they are also parametrized by its axiomatic definitions. Therefore, the user first needs to set a constant value for each axiomatic definition and then Fastest derives test cases considering those values. In this way, the application is tested for only one of its possible meanings.

6.10 Avoid Arbitrary Numeric Constants

If *memSize* stands for the amount of available memory of some computer and the specification includes an axiomatic definition such as:

$$memSize : \mathbb{N}$$
$$memSize = 1024$$

and an operation like:

WriteMemOk

$$\Delta MemoryState$$
$$x? : BYTE$$
$$\#mem < memSize$$
$$mem' = mem \frown \langle x? \rangle$$

then possible test classes cases may be:

- $mem = \langle\rangle$

- $\#mem = 1$
- $\#mem = memSize - 1$
- $\#mem = memSize$
- $\#mem = memSize + 1$

and possible corresponding abstract test cases may be:

- $x? = byte0 \land mem = \langle\rangle$
- $x? = byte0 \land mem = \langle byte0\rangle$
- $x? = byte0 \land mem = \langle byte0, \ldots, byte0\rangle$ where ... represents 1021 elements
- $x? = byte0 \land mem = \langle byte0, \ldots, byte0\rangle$ where ... represents 1022 elements
- $x? = byte0 \land mem = \langle byte0, \ldots, byte0\rangle$ where ... represents 1023 elements

It is very important to remark that, although we have used "..." to represent elements in each sequence, in the real abstract test cases the elements must be written down. Precisely, the difficulty in writing down those test cases is the reason for which we suggest avoiding arbitrary numeric constants. Fastest will automatically find the first two abstract test cases but it will need user assistance in order to find the remaining three.

However, if the model is slightly rewritten Fastest will automatically derive all the abstract test cases but one and, likely, the implementation will be verified as thouroghusly as with the other model. The only change is to avoid the constant by rewriting the axiomatic definition as follows:

$$\begin{array}{|l} memSize : \mathbb{N} \\ \hline 0 < memSize \end{array}$$

Then, the user can derive the same test classes and later he or she binds a smaller constant to $memSize$, 3 is an optimal choice. In this way, Fastest will automatically find the following test cases:

- $x? = byte0 \land mem = \langle\rangle$
- $x? = byte0 \land mem = \langle byte0\rangle$
- $x? = byte0 \land mem = \langle byte0, byte0\rangle$
- $x? = byte0 \land mem = \langle byte0, byte0, byte0\rangle$

and the user will have to assist the tool to find the remaining one:

- $x? = byte0 \land mem = \langle byte0, byte0, byte0, byte0\rangle$

If the implementation can be *configured* to think that the available memory is 3 bytes, then it is very likely that these test cases will uncover the same errors than the original ones—and perhaps in less time. It is important to remark that this will work if the implementation is configured, and not modified. In other words, it will work if there is some symbolic constant that can be modified without changing a single line of code or if this value is returned by some external function.

7 User's Manual

Throughout of this manual remember that Fastest is still a prototype. Then, it is not as robust as it should be.

For an academic presentation of Fastest see [CR09] and [CAR10].

7.1 Installing and Executing Fastest

Fastest should work on any environment with Java SE Runtime Environment 1.6 or newer. However, it has been tested only on Linux and MS-Windows boxes. To install the tool, just decompress and unarchive the file Fastest.tar.gz in any folder of your choice.

Fastest can run in one of two modes, we call them *application mode* and *distributed mode*. Roughly speaking, in application mode all the processing is performed in the local computer, while in distributed mode some tasks can be distributed across so called *testing server*. In this section we explain how to run Fastest in application mode; section 7.3 explains how to run it as a distributed system. It should be noted that, although application mode is the easiest way to use Fastest and it provides the same functionality than in distributed mode, Fastest achieves all of its efficiency in distributed mode.

7.1.1 Running Fastest and Entering Commands

To run Fastest in application mode, open a command window and run one of the following commands, where `INSTALLDIR` is the full path to the directory where Fastest was installed.

```
java -jar INSTALLDIR/fastest.jar
```

```
java -Xss8M -Xms512m -Xmx512m -jar INSTALLDIR/fastest.jar
```

The first command will serve for most purposes, but if large specifications will be used then the second command is a better option. If the computer has at least 1 Gb of memory then the second option should be used. The `Xms` and `Xmx` options indicate the minimum and maximum amounts of memory that the Java process will be able to use, respectively. Then, if more memory is needed increase the maximum (it must be a multiple of 1024). In this version of Fastest it is difficult to know if more memory is needed, but one symptom is command `genalltt` (section **??**) taking too long—more than one minute—to finish.

In either case, Fastest prints the following prompt from which users can issue commands.

```
Fastest>
```

To enter a command just type-in it along with its arguments and then press the return key. There are three ways of learning which commands are available:

- Type `help` and press the return key.
- Press the TAB key and a list of commands will be printed.
- Type-in the first letters of a command and then press the TAB key: either a list of the commands whose name starts with those letters will be printed, or the complete command name, if any, will be printed. For instance, if after entering the `a` key the TAB key is pressed the following is printed (`[TAB]` means pressing the TAB key):

```
Fastest>a[TAB]

addtactic apply
Fastest>a
```

And if the `d` key is pressed followed by the TAB key again, the result is the whole `addtactic` command printed, as follows:

```
Fastest>ad[TAB]dtactic
```

When the TAB key is pressed after command `loadspec`, Fastest prints the contents of the working directory. For example, if Fastest is run from the installation directory, the result is as follows:

```
Fastest>loadspec [TAB]

doc                    fastest-server.jar    fastest.jar
lib
Fastest>showsch
```

If the user types-in the first letters of one these files or directories and then presses the TAB key again, the name will be completed or a filtered list will be displayed, as with command names. If the letters correspond to a file name and it is completed, a blank space is added at the end; but if the letters correspond to a directory, when the name is completed a `/` or `\` character is added at the end. If the user presses the TAB key again, the content of this directory is displayed. The TAB key can be further pressed as a means of exploring the contents of the inner directories.

The left and right arrow keys can be used to move the cursor along the line being edited to modify it by inserting or deleting any character. The up and down arrow keys move across the commands that have been issued during the session. If one of these commands is recovered the user can modify it by using the left and right arrow keys, and can run it again by pressing the return key. Commands are executed when the return key is pressed regardless of where the cursor is.

Important!! Note that `Ctrl+C` kills the program making all the data and commands to be lost. Future versions will be more robust.

Important!! Fastest does not save anything by default. The user has to use one of the commands described in section 7.11 to save the data generated during a session.

7.2 Steps of a Testing Campaign

Roughly speaking, currently, a typical testing campaign carried on with Fastest can be decomposed in the steps listed below. Some of them are optional and some can be executed in a different order, as is described in the referred sections between brackets. Also, at any time users can run commands to explore the specification and the testing trees, and to save their results (7.11).

1. Install and declare more testing servers—i.e. run Fastest in distributed mode (7.3).
2. Load the specification (7.4).
3. Select the operations to be tested (7.4).
4. Select a list of testing tactics to be applied to each operation (7.5).

5. Generate testing trees (one for each selected operation) (**??**).
6. Set a value for each axiomatic definition (7.6.5).
7. Prune the trees (7.7.1).
8. Calculate abstract test cases (7.8).
9. If some leaves do not have an abstract test case, then explore these leaves to determine the cause for that. There are two possible causes:
   - The leaf predicate is a contradiction, but Fastest failed in pruning it. In this case:
     (a) Improve the elimination theorem library with as many elimination theorems as necessary to prune all the contradicting leaves (7.7.2).
     (b) Reload the elimination theorem library (**??**).
     (c) Prune the trees again (7.7.1).
     (d) Check whether all of the contradicting leaves have been pruned. If not, check the elimination theorems that were added and go to step 9a.
   - The leaf predicate is not a contradiction, but Fastest was not smart enough to find an abstract test case for it. In this case:
     (a) Help Fastest to generate finite models to find abstract test cases for these complex test classes (7.10).
     (b) Go to step 8.
10. If all of the leaves have an abstract test case, then save the results (7.11) and leave the program (7.12).

Step 4 is perhaps the most relevant step of all since it will determine how revealing and leafy testing trees are going to be. Along the same lines, step 7 is highly recommended since it will severely reduce the time of most testing campaigns.

Steps 4 and 5 can be executed iteratively and in the specified order or the opposite one.

Test case design includes from step 2 to step 5. The remaining steps generate test data, i.e. abstract test cases.

The following sections explain in detail each of the steps of a testing campaign carried on with Fastest.

7.3 Installing and Declaring Testing Servers

We have mentioned earlier that Fastest, by definition, is a distributed system making it possible to parallelize testing tree pruning and the calculation of abstract test cases, between all the available servers. Hence, a typical Fastest facility would consist of some clients, used by test engineers to conduct the testing, and many servers, used by those clients, to prune testing trees and calculate abstract test cases –it is important to remark that servers need not to be powerful computers, in fact we think of them being custom workstations which are usually underutilized. If test engineers use Fastest wisely then, they can take advantage of the servers during idle hours, optimizing the computing power of the organization.

In this version, testing servers need to be declared in each computer running a client. The configuration file named `lib/conf/cserversinfo.conf` is a text file that stores the list of available testing servers –for that particular client. The syntax is as follows:

```
name address port
```

where, `name` is a descriptive name for the server (it has no use in this version), `address` is the IP address of the server, and `port` is the port number at which the server is listening to.

On the other hand, each server has to be started either manually with the command `java -jar fastest-server.jar`; or it can be run by an operating system initialization script executing the same command. The port number where each server will listen for connections is declared in the configuration file `lib/conf/server-port.conf` by simply writing the number. Note that this number and the port number declared in the clients must coincide.

Fastest's default configuration is to run a client and a sever in the same computer. This is highly advisable since the client will be idle most of the time during a testing campaign. In the default configuration, client and server communicates through port 5001 while the IP address of the server must be 127.0.0.1.

7.4 Loading a Specification and Selecting Schemas

An specification is loaded by executing `loadspec` followed by a file name. The full path to the file must be written if it is not located in the directory from which Fastest was started, as in the following example:

```
Fastest> loadspec doc/sensors-simp.tex
```

It is assumed that the file is a text file containing the full specification; the current version does not support the LaTeX directive `\input{}`. If the specification contains syntactic or type errors it will not be loaded and the errors will be informed. It is possible to load only one specification at a time. To load a new specification run the `loadspec` command again or reset the current session by running command `reset` (in either case all the data generated so far will be lost).

It is possible to load specifications where terms are used before their declarations.

Once a specification has been loaded, it can be explored, printed and saved with the commands described in section 7.11.

After loading a specification the user has to select one or more schemas to be tested. Only schemas representing operations can be selected. A schema represents an operation if it contains any combination of the following: (a) an input or unprimed state variable, or (b) an output or primed state variable. To select an schema use `selop` followed by the name of a Z schema representing an operation. The list of candidate Z schemas can be displayed with `showloadedops`, with no arguments. It can be selected as many schemas as needed by issuing the same number of `selop` commands. A schema that was previously selected can be deselected with command `deselop` followed by its name.

7.5 Applying Testing Tactics and Generating Testing Trees

Testing tactics can be applied to any sub-tree of any (previously) selected schema—in particular they can be applied to the entire tree. To apply a testing tactic to a particular sub-tree, that sub-tree must already exist. Then, the first tactic can only be applied to the VIS of the operation. The first tactic applied by Fastest is always Disjunctive Normal Form (DNF, see below). To apply DNF to all the selected schemas just run `genalltt`.

Except for DNF, tactic application is performed in two steps:

1. Add the tactic to the list of tactics to be applied.
2. Run `genalltt`.

These steps can be repeated as many times as needed. They can also be interleaved with any command to prune testing trees (7.7), what is highly advisable. It is also possible to run these steps

even before `genalltt` is run to apply DNF, in which case Fastest first applies DNF and then the tactics added by the user—DNF is applied only once the first time `genalltt` is run. Testing trees can be displayed with `showtt` (7.11).

Important!! If `genalltt` takes more than a couple of minutes to finish it might be the case that the Java process run out of memory. It usually happens when the DNF of an operation has thousands of disjuncts—this, in turn, occurs when the operation is too complex considering full schema unfolding. If this occurs, the program will look like tilt—we hope to solve this in future versions. The only thing the user can do is to kill the process from the operating system. This problem might be solved by augmenting the memory available for the Java process (7.1).

The command `addtactic` adds a testing tactic to the list of tactics to be applied to a particular (previously selected) operation. Tactics are applied in the order they are entered by the user. Initially, the list of tactics of any operation includes only DNF, which is the first to be applied. The command syntax is rather complex because it depends on the tactic that is going to be applied (see the following sections for more details). The base syntax is:

```
addtactic sub_tree tactic_name parameters
```

where `sub_tree` is the name of either a selected schema or the name of a test class already generated, `tactic_name` is the name of a tactic supported by Fastest, and `parameters` is a list of parameters that depends on the tactic.

If `sub_tree` is the name of a schema, the tactic is applied to all the existing leaves of the corresponding testing tree. If `sub_tree` is the name of an existing test class, the tactic is applied to all the leaves of the sub-tree whose root node is that test class. The examples shown in Figure 14 may clarify this behaviour.

Unless `addtactic` prints an error message, the tactic has been successfully added. This command produces no other effect than adding the tactic to an internal list until command `genalltt` is executed.

Command `showtactics` prints a brief description of the available tactics; the following sections describe them in more detail.

7.5.1 Disjunctive Normal Form

This tactic is applied by default and it must not be selected with `addtactic`. By applying this tactic the operation is written in Disjunctive Normal Form and the *VIS* is divided in as many test classes as terms are in the resulting operation's predicate. The characteristic predicate of each class is the precondition of one of the terms in the operation's predicate.

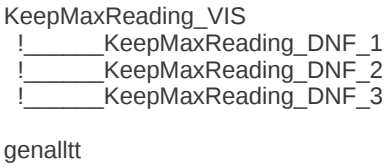
7.5.2 Standard Partition (Fastest's name SP)

This tactic uses a predefined partition of some mathematical operator (see "Standard domains for Z operators" at page 165 of Stocks' PhD thesis [Sto93]).

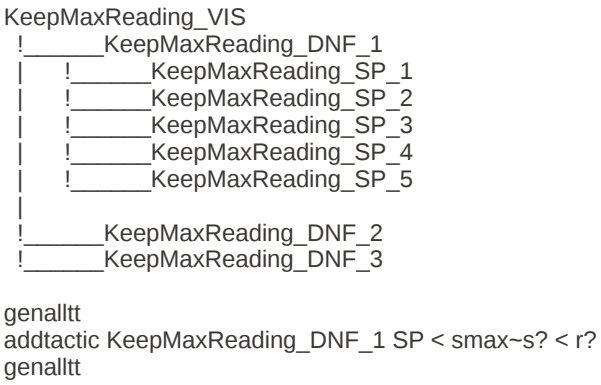
Take a look at Appendix C and at the file `INSTALLDIR/lib/conf/stdpartition.spf` to see what standard partitions are delivered with Fastest and how to define new ones. We think the syntax is rather straightforward. The user can edit this file to change, erase or add standard partitions, thus making this tactic quite powerful and flexible. Fastest needs to be restarted if this file is changed because it is loaded only during start up.

To apply one of those standard partitions to an operation the command is as follows.

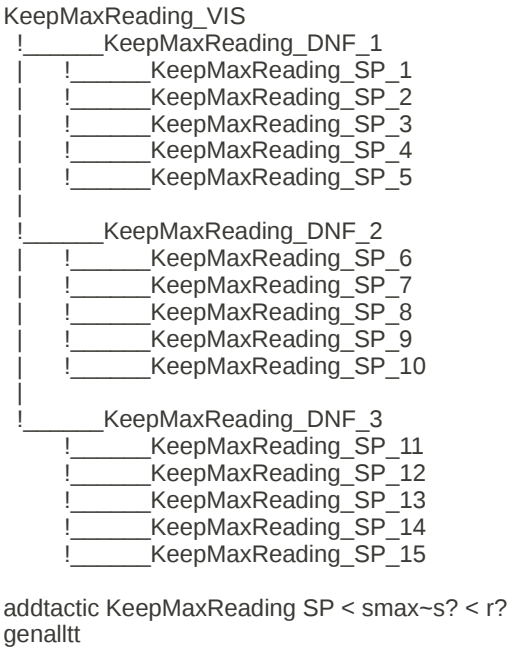
```
addtactic op_name SP operator expression
```



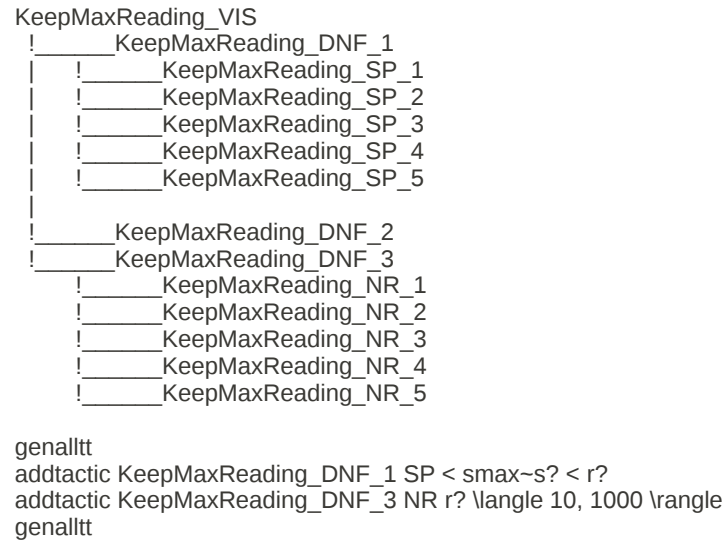
(a) Applying just DNF.



(b) Applying DNF and then SP to just one test class.



(c) Applying DNF and SP to the entire testing tree.



(d) Applying DNF and then two different tactics to two different test classes.

Figure 14: In each figure we show the testing tree produced with the script shown below them (scripts include only the relevant commands).

where `operator` is the LaTeX string of a Z operator and `expression` is a Z expression written in LaTeX. It is assumed that `operator` appears in the `expression` and this in turn appears in the predicate of the selected operation. Hence, this tactic can be applied to different operators and different expressions of the same operation.

The application of the tactic divides each test class at a given level of the testing tree in as many test classes as conjunctions defines the partition. Each conjunction is conjoined to the predicate of the test class being partitioned to form a new test class.

7.5.3 Free Type (Fastest's name FT)

This tactic generates as many test classes as elements a free type (enumerated) has. In other words if a model defines type $COLOUR ::= red \mid blue \mid green$ and some operation uses c of type $COLOUR$, then by applying this tactic each test class will by divided into three new test classes: one in which c equals red, the other in which c equals $blue$, and the third where c equals $green$.

The tactic is applied with the following command:

```
addtactic op_name FT variable
```

where `variable` is the name of a variable whose type is a free type.

Currently, Free Type works only if the free type is actually an *enumerated* type, i.e. an inductive type defined only by constants.

7.5.4 In Set Extension (Fastest's name ISE)

It applies to operations including predicates of the form $expr \in \{expr_1, \ldots, expr_n\}$. In this case, it generates n test classes such that $expr = expr_i$, for i in $1 \,..\, n$, are their characteristic predicates. The command to add this tactic is as follows:

```
addtactic op_name ISE predicate
```

where `predicate` is an atomic predicate of the form shown above.

7.5.5 Proper Subset of Set Extension (Fastest's name PSSE)

This tactic uses the same concept of ISE but applied to set inclusions. PSSE helps to test operations including predicates like $expr \subset \{expr_1, \ldots, expr_n\}$. When PSSE is applied it generates $2^n - 1$ test classes whose characteristic predicates are $expr = A_i$ with $i \in 1 \,..\, 2^n - 1$ and $A_i \in \mathbb{P}\{expr_1, \ldots, expr_n\} \setminus \{\{expr_1, \ldots, expr_n\}\}$. $\{expr_1, \ldots, expr_n\}$ is excluded from $\mathbb{P}\{expr_1, \ldots, expr_n\}$ because $expr$ is a proper subset of $\{expr_1, \ldots, expr_n\}$. The command syntax is as follows:

```
addtactic op_name PSSE predicate
```

where `predicate` is an atomic predicate of the form shown above.

7.5.6 Subset of Set Extension (Fastest's name SSE)

It is similar to PSSE but it applies to predicates of the form $expr \subseteq \{expr_1, \ldots, expr_n\}$ in which case it generates 2^n by considering also $\{expr_1, \ldots, expr_n\}$. The command syntax is as follows:

```
addtactic op_name SSE predicate
```

where `predicate` is an atomic predicate of the form shown above..

7.5.7 Numeric Ranges (Fastest's name NR)

With this tactic the user can bind an ordered list of numbers, $n_1, \ldots n_k$, to a numeric variable, *var*, in such a way that, when the tactic is applied, it generates $2 * k + 1$ test classes characterized by the following predicates: $var < n_1$, $var = n_1$, $n_1 < var < n_2$, ..., $var = n_i$, $n_i < var < n_{i+1}$, $var = n_{i+1}$, ..., $var < n_k$, $var = n_k$ and $n_k < var$. Consider the following example.

Variable appearing in operation	$memPointer : \mathbb{N}$
List provided by the user	$\langle 0, 65535 \rangle$
Test classes generated by the tactic	$T_1 \rightarrow memPointer < 0$ $T_2 \rightarrow memPointer = 0$ $T_3 \rightarrow 0 < memPointer \land memPointer < 65535$ $T_4 \rightarrow memPointer = 65535$ $T_5 \rightarrow 65535 < memPointer$

The command to apply this tactic is as follows:

```
addtactic op_name NR variable \langle list of numbers \rangle
```

where `variable` is the name of a numeric variable appearing in the operation; and each element in the list must be separated by a comma and in increasing order. The list must be non empty. If the type of the variable is $\mathbb{N}$, Fastest checks that all the numbers in the list are naturals.

Important!! In this version, Fastest will accept lists of numbers in any order but the behaviour of the tactic will be unpredictable.

7.5.8 Mandatory Test Set (Fastest's name MTS)

With this tactic the user can bind a set of constants, $\{v_1, \ldots, v_n\}$ to an expression, *expr*, in such a way that, when the tactic is applied, it generates $n + 1$ test classes characterized by the following predicates: $expr = v_i$ for all i in $1 \,..\, n$, and $expr \notin \{v_1, \ldots, v_n\}$.

The command to apply this tactic is as follows:

```
addtactic op_name MTS "expr" set_extension
```

where `expr` is an expression appearing in the operation and `set_extension` is a set extension, written in LaTeX mark-up, whose members are constants. Fastest checks whether the types of `expr` and `set_extension` are consistent.

In this version, the constants in `set_extension` can be numbers, elements of enumerated types, identifiers declared in axiomatic definitions or constants assembled out of them—for instance, $2 \mapsto ON$, where ON is an element of an enumerated type.

7.5.9 Weak Existential Quantifier (Fastest's name WEQ)

This tactic can be applied only when the precondition of the operation has one or more existential quantifications or negations of universal quantifications. It should be noted that this tactic might not produce a partition of the test classes where it is applied—if this is unacceptable, then see section 7.5.10.

Conceptually, this tactic transforms a quantification over a potentially infinite set into a quantification over a user-provided set extension. Since an existential quantification over a finite set is

equivalent to a disjunction, then WEQ first transforms the existential quantification into a disjunction. Then it writes the disjunction into DNF and finally it generates as many test classes as terms the DNF has plus one more characterized by the negation of the other predicates. Consider the following example.

Original predicate	$\exists\, x : \mathbb{N};\ y : \operatorname{seq}\mathbb{Z} \bullet x > w \land y \neq \langle\rangle \Rightarrow head\ y > x$
Set extensions provided by the user	$x == \{4, 6\}, y == \{\langle 4\rangle\}$
First transformation	$(4 > w \land \langle 4\rangle\ \neq \langle\rangle \Rightarrow head\ \langle 4\rangle\ > 4)$ $\lor\ (6 > w \land \langle 4\rangle\ \neq \langle\rangle \Rightarrow head\ \langle 4\rangle\ > 6)$
The predicate is written in DNF	$4 > w \land y = \langle\rangle$ $\lor\ 4 > w \land head\ \langle 4\rangle\ > 4$ $\lor\ 6 > w \land y = \langle\rangle$ $\lor\ 6 > w \land head\ \langle 4\rangle\ > 6$
Test classes generated by the tactic	$TC_1 \rightarrow 4 > w \land y = \langle\rangle$ $TC_2 \rightarrow 4 > w \land head\ \langle 4\rangle\ > 4$ $TC_3 \rightarrow 6 > w \land y = \langle\rangle$ $TC_4 \rightarrow 6 > w \land head\ \langle 4\rangle\ > 6$ $TC_5 \rightarrow \neg\ (\exists\, x : \mathbb{N};\ y : \operatorname{seq}\mathbb{Z} \mid$ $x \notin \{4, 6\} \land y \notin \{\langle 4\rangle\} \bullet$ $x > w \land y \neq \langle\rangle \Rightarrow head\ y > x)$

In order to apply WEQ the user has to indicate the quantified predicate and a set extension for each bound variable—or a set extension for each type of the bound variables. The command to add this tactic is as follows:

```
addtactic Op_name WEQ "quantification" "bindings"
```

where `quantification` is the quantified predicate as appears in the operation and `bindings` is a comma separated sequence of bindings. Each binding must have the following form:

```
var|type == set_extension
```

where `var` is one of the quantified variables, `type` is the type of one of the quantified variables and `set_extension` is a set extension, written in LaTeX mark-up, whose members are constants. Fastest checks that all the quantified variables are bound, directly through their names or indirectly through their types, to a set extension and that the types of the set extensions are consistent with respect to the quantified variables. For instance, the command corresponding to the example shown above is:

```
addtactic SomOp WEQ "\exists x: \nat; y: \seq \num @ x > w \land
y \neq \langle \rangle \implies head~y > x" "x == \{4, 6\}, y == \{\langle 4
\rangle\}"
```

In this version, the constants in any `set_extension` can be numbers, elements of enumerated types, identifiers declared in axiomatic definitions or constants assembled out of them—for instance, $2 \mapsto ON$, where ON is an element of an enumerated type.

Important!! Fastest will not be able to automatically derive abstract test cases from test classes whose characteristic predicates include quantifications. This tactic and SEQ help to reduce this limitation, besides providing a sound tool to increase the accuracy of testing at the presence of quantifications.

Important!! This tactic tends to produce more repetitive abstract test cases than SEQ.

7.5.10 Strong Existential Quantifier (Fastest's name SEQ)

This tactic is an stronger form of WEQ since it always generates a partition of the test classes where it is applied by conjoining the following predicate to the i^{th} test class produced by WEQ, except to the last one:

$$\neg\,(\exists\, x_1 : T_1, \ldots, x_n : T_n \mid x_1 \neq val_i^1 \wedge \ldots \wedge x_n \neq val_i^n \bullet P(x_1, \ldots, x_n))$$

where $x_1 : T_1, \ldots, x_n : T_n$ are the quantified variables and their types, $val_i^1, \ldots, val_i^n$ are the i^{th} combination of values taken from the set extensions provided by the user for each quantified variable, and P is the quantified predicate.

For instance, in the example shown for WEQ, SEQ generates the following test classes:

$$\begin{array}{lll}
TC_1 & \rightarrow & 4 > w \wedge y = \langle\rangle \\
 & & \wedge \neg\,(\exists\, x : \mathbb{N};\ y : \mathrm{seq}\,\mathbb{Z} \mid x \neq 4 \wedge y \neq \langle 4\rangle \bullet x > w \wedge y \neq \langle\rangle \Rightarrow head\ y > x) \\
TC_2 & \rightarrow & 4 > w \wedge head\ \langle 4\rangle\ > 4 \\
 & & \wedge \neg\,(\exists\, x : \mathbb{N};\ y : \mathrm{seq}\,\mathbb{Z} \mid x \neq 4 \wedge y \neq \langle 4\rangle \bullet x > w \wedge y \neq \langle\rangle \Rightarrow head\ y > x) \\
TC_3 & \rightarrow & 6 > w \wedge y = \langle\rangle \\
 & & \wedge \neg\,(\exists\, x : \mathbb{N};\ y : \mathrm{seq}\,\mathbb{Z} \mid x \neq 6 \wedge y \neq \langle 4\rangle \bullet x > w \wedge y \neq \langle\rangle \Rightarrow head\ y > x) \\
TC_4 & \rightarrow & 6 > w \wedge head\ \langle 4\rangle\ > 6 \\
 & & \wedge \neg\,(\exists\, x : \mathbb{N};\ y : \mathrm{seq}\,\mathbb{Z} \mid x \neq 6 \wedge y \neq \langle 4\rangle \bullet x > w \wedge y \neq \langle\rangle \Rightarrow head\ y > x) \\
TC_5 & \rightarrow & \neg\,(\exists\, x : \mathbb{N};\ y : \mathrm{seq}\,\mathbb{Z} \mid \\
 & & \qquad x \notin \{4, 6\} \wedge y \notin \{\langle 4\rangle\} \bullet \\
 & & \qquad x > w \wedge y \neq \langle\rangle \Rightarrow head\ y > x)
\end{array}$$

The command to add this tactic is the same than WEQ but changing `WEQ` for `SEQ`.

Important!! Fastest will not be able to automatically derive abstract test cases from test classes whose characteristic predicates include quantifications. This tactic and WEQ help to reduce this limitation, besides providing a sound tool to increase the accuracy of testing at the presence of quantifications.

Important!! This tactic tends to produce more test classes for which Fastest would be unable to automatically find abstract test cases, than WEQ.

7.6 Dealing with Axiomatic Definitions

According to the TTF and to the semantics of the Z notation, identifiers declared in axiomatic definitions neither are state variables nor input variables. However, since they do appear in operations, they are carried all the way down to test classes. Hence, when abstract test cases have to be derived from test classes it is necessary to bind a value for each identifier declared in an axiomatic definition, because otherwise there is no way to find a tuple of values satisfying the test class' predicate—in this sense, Fastest treats all these identifiers as model parameters. Precisely, if only test case design is needed then this step can be skipped.

$[CHAR, USER]$

$$
\begin{array}{|l}
control, blank : \mathbb{P}\, CHAR \\
null, tab, space : CHAR \\
Max, Mid, Min : \mathbb{Z} \\
root : USER;\ adm, audit : \mathbb{P}\, USER \\
asciiTbl : \mathbb{N} \pfun CHAR \\
\hline
null \in control \\
control \cap blank \neq \varnothing \\
Min = 34 \\
Max = 1000 + Min \\
blank = \{tab, space\} \\
root \in adm \\
adm = audit \cup \{root\} \\
\end{array}
$$

Figure 15: Examples of identifiers declared in an axiomatic definition.

Fastest classifies identifiers declared in axiomatic definitions in the following categories, treating each of them in a different way as is described below.

7.6.1 Basic Types

An identifier, $ident : T$ where T is a basic type, declared in an axiomatic definition is considered a constant. For example *null*, *tab*, *space* and *root* in Figure 15 are considered to be constants of their respective types. These constants are used when Fastest calculates abstract test cases (7.8). The user does not need to take any action for these identifiers.

7.6.2 Symbolic Constants

A identifier, $ident : T$ where T can be any type but a basic one, declared in an axiomatic definition is a *symbolic constant* if there is exactly one equality of the form $ident = cexpr$, where *cexpr* is a *constant expression*. A constant expression is any valid expression verifying any of the following:

- The expression is a number or an element of an enumerated type.
- The expression includes only symbolic constants, numbers, elements of enumerated or basic types and Z symbols.

For example, in Figure 15, *Min*, *Max* and *blank* are symbolic constants. *Mid* is not a symbolic constant because there is no equality defining a constant value for it; and *adm* is not a symbolic constant neither because $audit \cup \{root\}$ is not a constant expression.

Fastest automatically reduces any constant expression to its equivalent constant value, thus binding the corresponding symbolic constant to this value. In other words, the user does not need to take any action for these identifiers. The user can see the identifiers for which Fastest automatically bound a constant value by running command `showaxdefvalues`.

7.6.3 Equalities

If an identifier, $ident : T$ where T is any type, is declared in an axiomatic definition, and there is exactly one equality of the form $ident = expr$, where *expr* is not a constant expression, then

users should bind a value for each identifier in *expr* for which Fastest does not automatically bind a constant value and, at the same time, they cannot bind a value to *ident*. Once this is done, Fastest automatically reduces *expr* to a constant value, which is bound to *ident*. If this is not done, then Fastest will not be able to find abstract test cases for all test classes.

For example, users need to manually bind a value to *audit* in Figure 15 but they cannot bind a value to *adm*.

The user can bind values to identifiers with command `setaxdef` (7.6.5).

7.6.4 All Other Declarations

Identifiers declared in axiomatic definitions that do not meet the conditions described in the previous sections fall in this category[5]. For these identifiers the user should give constant values so Fastest has chances to find abstract test cases for all the test classes.

The user can bind values to identifiers with command `setaxdef` (7.6.5).

7.6.5 Command `setaxdef`

Fastest provides command `setaxdef` to bind values to identifiers declared in axiomatic definitions—provided they can be bind at all. Command syntax is as follows:

```
setaxdef ident ["constant_declarations"] "value"
```

where, `ident` is the identifier for which the user wants to set a constant value and `value` is that value. This means that Fastest will replace the identifier for the value when reducing expressions (7.6.3) and when calculating abstract test cases. The optional parameter `constant_declarations` must be used when the `value` refers to constants of basic types (see an example below). For example, the following command sets a value for *Mid* (declared in Figure 15):

```
setaxdef Mid "517"
```

When such a command is issued, Fastest checks that the type of the value is consistent with the type of the identifier. Also, it *tries* to check that the value satisfies all of the predicates, appearing in axiomatic definitions, where the identifier is referenced. However, this check can only be finished when all these predicates become constant, i.e. when all the variables have been bound to a constant value. Then, when the last identifier is bound to a constant, the predicate is evaluated and, possibly, an error message is printed. Therefore, if Fastest complains that the value that was last bound to an identifier does not verifies a predicate in an axiomatic definition, the user should check whether this last value is the cause of the problem or it is the values previously bound to the other identifiers. If this is the case, the user can reset the previous values with the same command, until no error messages are printed. Command `eval` (**??**) might be useful to check whether values satisfy predicates or not.

Now, let's see an example involving the optional parameter `constant_declarations`. Say *asciiTbl* (defined in figure 15) is used in some operation. Then, Fastest needs that the user sets a value for it so the tool can find abstract test cases for all the test classes generated for the operation. In the same axiomatic definition have been defined some *CHAR*'s but, say, the user wants to test the operation with a more realistic ASCII table. Hence, the user can issue a command like this one:

```
setaxdef ident "char0,char1,char2:CHAR"
               "\{0 \mapsto null, 1 \mapsto char0, 2 \mapsto char1,
                  3 \mapsto char2, 4 \mapsto tab, 5 \mapsto space\}"
```

[5]We will further subdivide this category to solve some issues automatically in future releases.

In other words, `constant_declarations` allows the user to declare some constants of basic types that are used to define the constant value to be bound to the identifier. Internally, Fastest declare these identifiers in axiomatic definitions. Although the user can chose any names in the declaration, there are two things worth to mention:

1. Avoid name clashes with other identifiers declared in axiomatic definitions and operations.
2. Chose names that increase the likeness of Fastest finding abstract test cases by following the rules described in section 7.10.1.

If constants of different types need to be declared, the syntax is the same than in Z, i.e.:

```
setaxdef ident "char0,char1,char2:CHAR; user0,user7:USER" ...
```

The user can see the values bound to identifiers by running command `showaxdefvalues`. Besides, Fastest provides command `showaxdefs` so users can easily see all the axiomatic definitions used in the specification.

`setaxdef` can be executed right after `loadspec`, but perhaps the best moment to do it is after `genalltt`. Fastest will not be able to find abstract test cases for those test classes where an identifier, declared in an axiomatic definition, is referenced and there is no constant value bound to it. Therefore, if abstract test cases are needed for all the test classes, `setaxdef` must be executed before `genalltca`. However, both commands can be executed one after another iteratively until there is an abstract test cases for each test class.

7.7 Pruning Testing Trees

It is very common that many leaves in a testing tree are indeed empty classes because their predicates are contradictions. Finding an abstract test case for a given test class implies finding a vector of constants verifying the predicate of the test class. Hence, if the predicate is a contradiction, it will be impossible to find an abstract test case for it. Fastest provides two strategies to prune empty test classes from testing trees. The following sections describe each of them.

7.7.1 Automatic Pruning

Fastest provides command `prunett` which goes through all the testing trees and prunes as many empty test classes as it can. In general, `prunett` will not be able to prune all the empty test classes since this problem is undecidable. Then, once `prunett` finishes, it is possible that some empty test classes remain hanging from the tree. In this case the engineer has two alternatives—we recommend to follow the first one:

1. Run `genalltca` to find abstract test cases from the remaining leaves (7.8). If Fastest could not find abstract test cases for some of them, then expand each of these leaves and analyze each of them in order to determine whether they are contradictions or not. If some are, the engineer has, again, two options that will be described below; if not, `setfinitemodel` should be used (7.10).
2. Expand each leaf and analyze it in order to determine whether is a contradiction or not. If some are, the engineer has, again, two options that will be described below; if not, run `genalltca` (7.8).

In any case the engineer has two options when there are empty test classes that were not pruned by `prunett`. We strongly recommend to follow the first one:

1. Add one or more *elimination theorems* (7.7.2), then run `loadelimtheorems` (7.7.3) and finally run `prunett` again.

2. Prune the test classes manually (7.7.5).

If all the leaves of a given test class were pruned, `prunett` will try to prune that test class too. `prunett` and `genalltca` (7.8) can be run iteratively: the former will try to prune only those leaves for which no abstract test case was found, and the latter will try to find an abstract test case for them.

`prunett` is quite fast but if the number of test classes is very large or they have large predicates, the process might take several minutes.

7.7.2 Elimination Theorems

A test class should be pruned from a testing tree when it is an empty set. A test class is an empty set when its predicate is a contradiction. For instance, the following test classes are empty sets since in *SCAddCat_SP_1* the proposition $\{c?\} = \{\}$ is false, and in *SCAddCat_SP_5* the conjunction $categs \neq \{\} \land categs \subset \{c?\}$ is a contradiction.

SCAddCat_SP_1

$level : \mathbb{Z}$
$categs : \mathbb{P}\, CATEGORY$
$MAXNCAT : \mathbb{N}$
$c? : CATEGORY$

$c? \notin categs$
$categs = \{\}$
$\{c?\} = \{\}$

SCAddCat_SP_5

$level : \mathbb{Z}$
$categs : \mathbb{P}\, CATEGORY$
$MAXNCAT : \mathbb{N}$
$c? : CATEGORY$

$c? \notin categs$
$categs \neq \{\}$
$\{c?\} \neq \{\}$
$categs \subset \{c?\}$

`prunett` analyses the predicate of each leaf in a testing tree to determine if the predicate is a contradiction or not. Since the problem of finding contradictions in first order logic is undecidable, Fastest rests on a best-effort algorithm. The algorithm uses a library of so called *elimination theorems* each of which represents a contradiction. For example the following two elimination theorems are included in the library:

Elimination Theorem SingletonIsNotEmpty $[x : X]$

$\{x\} = \{\}$

Elimination Theorem NotSubsetOfSingleton $[A : \mathbb{P}\, X;\ x : X]$

$A \neq \{\}$
$A \subset \{x\}$

Fastest uses the library of elimination theorems to prune inconsistent test classes. If it finds that part of a test class matches on of the elimination theorems, then it prunes that test class from the tree—remember that the predicate of a test class is a conjunction of atomic predicates, so if one or more of them are contradictory, then the class is inconsistent. Hence, the more elimination theorems in the library, the more test classes will be pruned by `prunett` without user intervention. We suggest that users should add elimination theorems to the library every time Fastest fails in pruning an empty test class (instead of pruning it manually) due to two reasons:

1. More test classes could be pruned automatically in the current or future projects; and

2. It enables the chances to formally prove that the elimination theorem is actually a contradiction.

As the reader can see, when executing `prunett`, it displays either the name and parameters of the elimination theorem by means of which the test class was pruned, or it says that Fastest cannot prune the test class—which, as we have seen, does not mean that the test class cannot be pruned at all.

The library of elimination theorems is the text files `INSTALLDIR/lib/conf/elimTheorems.tex` and `INSTALLDIR/lib/conf/rwRules.tex`. The user can add, delete and modify elimination theorems by editing the first file with any text editor. The syntax of the elimination theorems is rather easy an can be learned by reading the library; here we just give some tips. Please read all them carefully before modifying the library.

For a more academic presentation of the pruning method implemented by Fastest read [CAR10].

LaTeX. Elimination theorems are written in LaTeX using the CZT package.

Theorem names. Each elimination theorem must have a unique name.

Formal parameters. Each elimination theorem has a set of formal parameters. The parameters can be any Z legal declaration of variables plus the reserved word `\const` before a parameter name. If a parameter is preceded by `\const` it means that Fastest will replace it only by constants of the corresponding type. `\const` applies only to parameters of type $\mathbb{Z}$, $\mathbb{N}$ or any enumerated type (i.e. free types without recursion). When an elimination theorem contains two or more constant parameters, they are replaced only by different literals. For instance, the library contains the following elimination theorem.

Elimination Theorem ExcludedMiddleEq $[x, \text{const } y, \text{const } z : X]$

$x = y$
$x = z$

Fastest will "call" this elimination theorem always with $y \neq z$; for example, ExcludedMiddleEq (n, 11,34), but never something like ExcludedMiddleEq(n,34,34) nor ExcludedMiddleEq(n,*count*,3).

The parameters of an elimination theorem are formal in the sense that Fastest will try to replace them with the actual types of the terms appearing in the predicate of a test class. Hence, by using adequate parameters the theorem will serve to prune a lot of test classes. See the paragraphs **Rewrite rules**, **Subtyping substitutions** and **Syntactic substitutions** below for other mechanisms that enable the generalization of elimination theorems.

The body of theorems. The predicate of an elimination theorem must be a conjunction of atomic predicates. The conjunction must be written only one conjunct by line, ending each line with `\\`.

Since the predicate is a conjunction the order of the conjuncts is unimportant.

An atomic predicate in an elimination theorem can be any legal Z atomic predicate using the standard symbols of Z supported by Fastest, the names of the formal parameters and the reserved words `\sw`, `\anything`, `\se` and `\eval`, explained below.

Somewhere. `\sw` takes a Z LaTeX string enclosed in brackets and separated by one blank from each bracket. For instance, the library contains the following elimination theorem:

Elimination Theorem BasicUndefinition $[f : X \pfun Y;\ x : X]$

$x \notin \operatorname{dom} f$
somewhere($f\ x$)

`\sw( string )` is equivalent to the regular expression `*string*`. When `prunett` finds such a directive it tries to match the regular expression in any of the atomic predicates of a test class' predicate.

Anything. `\anything` takes no parameters. For example the following elimination theorem uses this directive.

Elimination Theorem SetExtNotASeq $[s : \operatorname{seq} X;\ n : \mathbb{N}]$

$n = 0$
$s \neq \{\}$
$\operatorname{dom} s = \operatorname{dom}\{i : 1 \mathinner{..} \text{anything} \bullet i + n - 1 \mapsto \text{anything}\}$

`\anything` is equivalent to the regular expression that matches any string (`*`). Two or more occurrences of this directive can match different strings.

Set extensions. `\se` takes a Z LaTeX string enclosed in brackets and separated by one blank from each bracket. The string is intended to be an element of a set extension. For instance, the library contains the following elimination theorem:

Elimination Theorem SingletonNotSet $[A : \mathbb{P} X;\ x : X]$

$x \notin A$
$\text{setExtension}(x) = A$

If `\se( expr )` is found in an elimination theorem, Fastest will try to match `expr` against any of the elements of a set extension present in the test class being processed. `expr` cannot contain commas; instead `\mapsto` must be used. If the members of the set extension present in the test class are set comprehensions, `\se` will not be useful—i.e. the expression will not match against the set comprehensions.

Then, if a test class contains any of the following conjuncts it will be pruned by SingletonNotSet:

$x \notin mem$
$\{x\} = mem$

$3 \notin A \cup B$
$\{1, 2, 3\} = A \cup B$

$R = \{b \mapsto 4, c \mapsto 26\}$
$b \mapsto 4 \notin R$

but Fastest will fail in pruning a test class including the following conjuncts, although it is a contradiction:

$(1, 3) \notin A \cup B$
$\{(1, 3), (2, 17)\} = A \cup B$

Evaluations. `\eval` takes a constant Boolean expression as a parameter separated by one blank from each bracket. A constant Boolean expression is a Boolean expression referencing parameters preceded by `\const`, $\mathbb{Z}$ literals, Z operators and the literals of enumerated types. The following elimination theorem uses it:

Elimination Theorem DomSizeIsZero $[R : X \leftrightarrow Y;\ \text{const}\ N, m : \mathbb{N}]$

Integers	$n < m$	$m > n$
	$n > m$	$m < n$
	$n \leq m$	$m \geq n$
	$n \geq m$	$m \leq n$
Sets	$A \cap B$	$B \cap A$
	$A \cup B$	$B \cup A$
All types	$x = y$	$y = x$
	$x \neq y$	$y \neq x$

(a) Equivalence rules.

Type	**Subtype**
$X \leftrightarrow Y$	$X \leftrightarrow Y$, $X \nrightarrow Y$, $X \rightarrow Y$, seq Y
$X \nrightarrow Y$	$X \nrightarrow Y$, $X \rightarrow Y$, seq Y
$\mathbb{Z}$	$\mathbb{Z}$, $\mathbb{N}$

(b) Subtyping rules.

Figure 16: Fastest applies by default these equivalence and subtyping rules.

$\# \operatorname{dom} R = m$
$m = N$
$R = \{\}$
$\text{eval}(N > 0)$

This sentence evaluates the parameter and if it is true and all of the other conjuncts of the theorem are found in the test class, then the test class is pruned. If the Boolean expression evaluates to false, the test class is not pruned.

Equivalence Rules Fastest applies equivalence rules, taken from the library, to the elimination theorems whenever possible. Besides, it applies by default the rules listed in Table 16a. This implies, for instance, that the engineer does not need to write the following theorem:

Elimination Theorem NotInEmptyDom_Silly $[x : X;\ R : X \leftrightarrow Y]$
$x \in \operatorname{dom} R$
$\operatorname{dom} R = \{\}$

because the library already contains the following one:

Elimination Theorem NotInEmptyDom $[x : X;\ R : X \leftrightarrow Y]$
$x \in \operatorname{dom} R$
$R = \{\}$

and the equivalence rule $R = \{\} \Leftrightarrow \operatorname{dom} R = \{\}$. Equivalence rules are lists of atomic predicates which should be equivalent to each other.

To add, delete or modify equivalence rules edit the file `INSTALLDIR/lib/conf/rwRules.tex` with any text editor.

Subtyping Rules Fastest also applies some simple subtyping rules when it substitutes the formal parameters of an elimination theorem or equivalence rule by actual parameters. A subtyping rule determines whether a type or set is a subtype of another type or set. For instance $X \rightarrow Y$ is a subtype of $X \nrightarrow Y$ which in turn is a subtype of $X \leftrightarrow Y$. The subtyping rules applied by Fastest are listed in Table 16b.

During the pruning process, Fastest substitutes formal parameters of type T in an elimination theorem for terms of any of the subtypes of T. Hence, if in a test class there is a term f of, say, "type" $\mathbb{N} \nrightarrow CHAR$, then the parameter R in theorem NotInEmptyDom above will be substituted by f because $X \nrightarrow Y$ is a subtype of $X \leftrightarrow Y$, $\mathbb{N}$ matches X and $CHAR$ matches Y. So, whenever the engineer adds an elimination theorem he or she should make his or her best effort in writing the most general possible formal parameters because the theorem will serve to prune more test classes in future projects.

Syntactic substitutions. Fastest also performs a number of syntactic substitutions that are common to the Z notation in the elimination theorems, rewrite rules and test classes. To name some, unnecessary parenthesis are removed, and expressions like `f~x`, `f(x)` and `f\ x` are written in a common format. In this way the user does not need to worry about how predicates have been written.

7.7.3 Loading the Elimination Theorem Library

The elimination theorem library is automatically loaded during start up. If the user adds or modifies the configuration file containing the library, these modifications will not have any effect until the library is loaded again. To do this, execute command `loadelimtheorems`.

7.7.4 Semiautomatic Pruning

Fastest provides two commands that can be used to prune empty test classes interactively.

- `searchtheorems class_name`, searches the library for elimination theorems that might be used to prune test class `class_name`.
- `apply theorem_name class_name "parameter" ... "parameter"`, applies the elimination theorem named `theorem_name` to the test class `class_name` to see if it is possible to prune the test class with that elimination theorem. The user has to provide a list of parameters enclosed between double quotes. If the list does not match either the number or the types of the formal parameters of the elimination theorem, then the test class will not be pruned. If the parameters provided by the user can be passed to the elimination theorem and the test class contains all the atomic predicates of it, then the test class is pruned. All the equivalence and subtyping rules are applied as with `prunett`.

  For instance the following commands prune test classes *SCAddCat_SP_*1 and *SCAddCat_SP_*5 shown at the beginning of the section.

```
apply SingletonIsNotEmpty SCAddCat_SP_1 "c?"
apply NotSubsetOfSingleton SCAddCat_SP_5 "categs" "c?"
```

7.7.5 Manual Pruning

Test classes can be pruned not only because they are empty, but also because they will not give meaningful test cases. This is at the engineer discretion. Fastest provides commands `prunefrom` and `prunebelow` to erase a sub-tree from some testing tree; and command `unprune` to restore previously erased sub-trees. Their syntax and semantics are as follows.

- `prunefrom class_name`

  This command deletes the sub-tree hanging from and including `class_name`. It is useful to erase leaves.

- `prunebelow class_name`

  This command deletes the sub-tree hanging from but not including `class_name`.

- `unprune class_name`

  This command restores the sub-tree hanging from and including `class_name`. Note that it is impossible to restore a sub-tree hanging from a pruned test class.

It is important to remark that pruning test classes from a testing tree manually can reduce the quality of testing. This happens when the tester prunes one or more classes that are not empty, and thus Fastest will not generate abstract test cases for them.

7.8 Generating Abstract Test Cases

`genalltca` distributes evenly the task of finding abstract test cases for test classes across all the registered testing servers (7.3). Then, the more testing servers the less the time needed to complete this task.

Important!! This process might take some hours. The tool will remain useless until the whole process terminates. If the process is interrupted, it will have to be restarted from the very begining. In that case all the abstract test cases generated so far will be lost.

This command can be run only after `genalltt` but it can be run as many times as needed, specially after `prunett` and `setfinitemodel` (7.10). Every time this command is run it will try to find an abstract test case for those leaves for which there is none. Then, if the user manages to prune more test classes or helps Fastest to find abstract test cases for the remaining classes, `genalltca` will run much faster than in previous executions.

Besides generating abstract test cases, the output of this command is a series of messages printed on the screen. For each test class being analysed, the command first prints a message like this one:

```
Trying to generate a test cases for the class: <tcn>
```

where `tcn` is the name of the test class. After some time Fastest will print one of the following messages for each test class:

- If Fastest found an abstract test case for a given test class, the following message is written:

  ```
  <tcn> test case generation -> SUCCESS.
  ```

- If Fastest was unable to find an abstract test case it will print one of the following messages (which one will be printed depends on the value of some configuration variables (**??**):

  ```
  <tcn> test case generation -> FAILED.
  ```

  ```
  <tcn> test case generation -> FAILED (without performing all the
  possible evaluations).
  ```

Once `genlltca` finishes, the user can explore and save the abstract test cases with command `showsch -tca` (7.11). Then, the user has to analyse those test classes for which a `FAILED` message was printed. As was explained above, the `FAILED` message might correspond to an empty test class that Fastest could not prune; in this case the user should proceed as suggested in section 7.7.1. If the test class is not empty, the the user should proceed as indicated in section 7.10.

7.9 Controlling the Generation of Abstract Test Cases

There are two configuration variables, defined in the file `INSTALLDIR/lib/conf/fastest.conf`, that can be used to fine tune the test case generation algorithm:

`MAX_EVAL` This is the maximum number of attempts to find one abstract test case for each test class, that the algorithm will perform. The larger this value, the higher the chances to find an abstract test case for each test class, but the longer it will take.

`DEF_SIZE_FINITE_SET` This is the maximum number of elements that will populate the finite sets bound to the most elemental types and variables. As before, the larger this value, the higher the chances to find an abstract test case for each test class, but the longer it will take. A small increment in this value might produce a really huge model where abstract test cases are sought. However, `MAX_EVAL` will limit the number of elements in this model that are tried out in order to find an abstract test case.

Every time these values are changed, Fastest must be restarted.

Fastest is delivered with the best values according to our experience.

When the finite model for a given test class is inspected up to `MAX_EVAL` elements but it has more, then the second message shown above is printed. This might be an indication that Fastest could have found an abstract test case for the test class if `MAX_EVAL` would have been larger.

7.10 Defining Finite Models

Finding an abstract test case for a given test class means to find a vector of constants that, when substituted by the free variables in the predicate, they satisfy it. This is an undecidable problem as the elimination of inconsistent test classes. Hence, Fastest may fail in finding an abstract test case due to three reasons:

1. The test class is an empty set—`prunett` failed because the problem it tries to solve is undecidable too.
2. Fastest is not "smart" enough to find one.
3. The search space is so large that Fastest aborted before exhausting it.

If the test class is empty, then proceed as indicated in section 7.7.1. If the test class is not empty, then keep reading this section.

Fastest calculates abstract test cases by generating a finite model for each leaf test class. This finite model is the Cartesian product between one finite set for each variable appearing in the VIS of the corresponding testing tree. Clearly, the bigger the finite model the more chances to find an abstract test case, but the more time it will take. Fastest automatically selects a finite model for each test class according to some heuristics. The experiments we have run so far show that the default strategy applied by Fastest discovers, in average, 90% of the abstract test cases of **non empty** test classes for moderated-size specifications. In complex models, however, the computing time needed to hit these percentages use to be quite high and can even be higher if testing trees are not properly pruned. Furthermore, in some cases the size of the default finite models grows exponentially reaching thousands of millions of elements. This could make Fastest to take years to find one abstract test case, *if the test class does not happen to be empty.* Hence, an efficient and effective pruning method is as important as giving the user the chance to help Fastest to select the most promising and smallest finite model.

Fastest allows the user to define a finite model at the test class level; in other words, a different finite model for each test class can be defined. If the user do not take any explicit action, Fastest generates a default finite model that we believe is the best option in most situations.

Important!! The user defined strategy should be used only as an exception; the default strategy works well in most models. The user defined strategy should be used only when a test class' **predicate** has many variables or some variables of complex types—such as higher order (partial) functions, relations between three or more sets, (partial) functions whose domain is a Cartesian product, and so on.

7.10.1 Default Finite Models

Finite models are constructed recursively from specific finite sets selected for all basic and enumerated types, and $\mathbb{N}$ and $\mathbb{Z}$, as follows:

- The default sets for basic types contain constants automatically generated by Fastest. If the identifier of a basic type is XYZ then the constants for that type will be $xyz0$, $xyz1$, $xyz2$ and all the identifiers of that type declared in axiomatic definitions. Fastest assumes that all these values are distinct from each other. For example, if the specification declares type $NAME$, and assuming there are no identifiers of this type declared in axiomatic definitions, then the default finite set for it will be $\{name0, name1, name2\}$, where it is assumed that $name0$, $name1$ and $name2$ are declared in some axiomatic definition and, since they have different names, they are all distinct from each other.

- The default sets for enumerated types are the types themselves.

- The default sets for $\mathbb{N}$ and $\mathbb{Z}$ depend on the predicate being evaluated. If some specific numbers appear in the predicate, then the finite sets will contain all of them and the minimum minus one and the maximum plus one. For instance, if a predicate references 0 and 42, then the finite set for $\mathbb{N}$ will be $\{0, 42, 43\}$—since $-1 \notin \mathbb{N}$ then it cannot be included in the set—and $\{-1, 0, 42, 43\}$ for $\mathbb{Z}$. If there are no such constants then the sets are $\{0, 1, 2\}$ and $\{-1, 0, 1\}$, for $\mathbb{N}$ and $\mathbb{Z}$ respectively.

- The finite sets for structured types (like functions, relations, power sets, etc.) are generated from the finite sets considered for the more basic types, essentially by making all the legal combinations between the elements of the finite sets defined for the basic types. For instance, the finite set for $\mathbb{P}\,\mathbb{Z}$ will be $\{\{\}, \{-1\}, \{0\}, \{1\}, \{-1, 0\}, \{-1, 1\}, \{0, 1\}, \{-1, 0, 1\}\}$ if the finite set for $\mathbb{Z}$ is $\{-1, 0, 1\}$.

7.10.2 Command `setfinitemodel`

If the default finite model does not satisfies the predicate of the test class being analysed—i.e. a `FAILED` message was printed for it—or if the finite model is too large—i.e. an `UNKNOWN` message was printed—the user can help Fastest to select a better finite model for that test class. The command to do this is as follows:

```
setfinitemodel class_name options
```

where `class_name` is the name of a test class in some testing tree and `options` can be any combination of:

- `-nzsize <size>`

  This option sets the size of the finite sets for $\mathbb{N}$ and $\mathbb{Z}$, if the class does not contain arithmetic constants; otherwise `Given` is applied –see below.

- `-btsize <size>`

  This option sets the size of the finite sets for all the basic types.

- `-size <size>`

  This option sets the size of the finite sets for all the preceding types. If either `nsize` and `size` or `btsize` and `size` are used together then `nsize` and `btsize` always takes precedence.

- `-fm "bindings"`

  where `bindings` is a semicolon separated sequence of one or more bindings. Each binding must have the following form:

  `var|type == set_extension`

  where `var` and `type` are any variable or type appearing in the VIS of the test class and `set_extension` is one of:

  - `a \upto b`

    This form can be used only for $\mathbb{N}$ and $\mathbb{Z}$. It restricts the finite set for the type or variable to the set $a \mathinner{..} b$ where a and b must be constant values.

  - A set extension, written in LaTeX mark-up, whose members are constants.

    It restricts the finite set for the indicated variable or type to the elements listed in the set extension.

    Note that if, for instance, `type` is $\mathbb{N} \pfun NAME$ then each element belonging to the set extension must be a (constant) partial function of type $\mathbb{N} \pfun NAME$ written in LaTeX mark-up. For instance, the following is a possible value for this case

    ```
    \{\{0 \mapsto name0, 1 \mapsto name1, 2 \mapsto name2\},
      \{0 \mapsto name0, 1 \mapsto name0, 2 \mapsto name0\}\}
    ```

    Note that the constants for type $NAME$ (i.e. $name0$, $name1$ and $name2$) are consistent with the constants automatically generated by Fastest for the same type. However, this is not mandatory and the user can chose any names as long as they, combined with those automatically generated by Fastest, satisfy the predicate of the test class.

  - `Given | Seeds`

    This form can be used only for $\mathbb{N}$ and $\mathbb{Z}$, but not for variables of those types. In this case `nsize` will not be considered. If this option is used the finite set for $\mathbb{N}$ or $\mathbb{Z}$ will be the set defined as follows:

    `Given`

    If this option is chosen, then Fastest will search for constants of type $\mathbb{Z}$ or $\mathbb{N}$ present in the test class, depending on the indicated variable or type. If there are no such constants, then $\{0, 1, 2\}$ or $\{-1, 0, 1\}$ are chosen depending on the indicated variable or type.

    If there are such constants, then a set of type $\mathbb{Z}$ will include all of them plus the integer number one unit less than the minimum of these constants and the integer number one unit more than the maximum of these constants. While a set of type $\mathbb{N}$ will have the same property changing "integer" by "natural" and noting that if zero is one of the constants then it will be the minimum.

    As has been said before, these are the default rules that are applied if the user does not run any `setfinitemodel` command for a test class.

    `Seeds`

    This option is similar to `Given`. It takes the same constants, adds the same upper and lower limits but also adds the mean value between each pair of consecutive constants found in the test class.

    Same considerations than in `Given` apply if there are no $\mathbb{N}$ or $\mathbb{Z}$ constants and if zero is one of the natural numbers found in the class.

As can be seen, this option sets the finite sets for the indicated variables or types.

If a definition is given for type T and another definition is given for variable x of type T, then the last takes precedence. Also, if there is another variable y of type T, for which no finite set was defined by the user, then the finite set for it will be the one defined for T.

In order to define the bindings for the `fm` option, command `eval` (`??`) might be helpful.

If one of the size options is used but there is a definition, within the `fm` option, for a type or a variable which involves a set with a different number of elements, then the last takes precedence. For example in `setfinitemodel any_class -nsize 3 -fm "\nat == \{1,2,3,4\}"`, the finite set for type $\mathbb{N}$ will be $\{1, 2, 3, 4\}$ although it has four elements, and not three. But if there are more than one variable of, say, type $\mathbb{N}$, one of which is x, then `setfinitemodel any_class -nsize 3 -fm "x == \{1,2,3,4\}"` will make Fastest to assign $\{1, 2, 3, 4\}$ to x and another three element set for the other variables, which will be calculated following the default rules.

`setfinitemodel` can be run after `genalltt` and before and after `genalltca`.

7.11 Exploring and Saving the Results

The specification and the results of the work carried on with Fastest can be displayed or saved in LaTeX format with the commands of the `show` family.

Important!! If Fastest terminates by any means, all data will be lost unless the user has saved it in files with one or more of the commands described in this section.

`showloadedops` prints the names of all the Z schemas that look like operation schemas. Fastest considers that a Z schema is an operation schema if it includes input or before state variables, on one hand, and output or after state variables, on the other. If an schema is the result of an schema expression, then all of them might be considered operations. For instance, if $A == B \lor C$ and B and C are operation schemas, then `showloadedops` will print something like:

```
* A
* B
* C
```

However, the user should select only A as an operation to be tested since B and C will be considered when DNF is applied. Operation selection is explained in section 7.4.

`showtatics` prints a brief description of all the available testing tactics. A deeper explanation of them can be fund in section 7.5 and its subsections.

The remaining `show` commands display and save the specification, testing trees, test classes, abstract test cases and values bound to identifiers declared in axiomatic definitions. In any case, command options must be entered in the order they are documented. Some commands feature the `-o` option that redirects the output to a file. This is the only way, so far, to save the results generated by Fastest. The output of most of these commands is LaTeX mark-up. The following table summarizes these commands.

Command	Description	Options
`showaxdefs`	Displays the axiomatic definitions present in the specification in LaTeX mark-up.	`[-o <file_name>]` redirects the output to a file
`showaxdefvalues`	Displays the values bound, either automatically or manually, to identifiers declared in axiomatic definitions.	`[-o <file_name>]` Same as before.

Command	Description	Options

`showsch`	Displays a given schema (it can be either any of the specification schema, test classes or abstract test cases) LaTeX mark-up.	`<sch_name>`
		The name of the schema to be displayed.
		`[-u <unfold_order>]`
		Displays the result with more or less detail (basically it expands up to some level the included schema boxes). `-u -1` expands all the schemas.
		`[-o <file_name>]`
		Same as before.

`showsch -tca`	Displays all of the schemas corresponding to abstract test cases (of all testing trees) LaTeX mark-up.	`[-p <op_name>]`
		Displays only the abstract test cases of operation schema `op_name`.
		`[-u <unfold_order>]`
		Same as before.
		`[-o <file_name>]`
		Same as before.

`showsch -tcl`	Displays all of the schemas corresponding to test classes (of all testing trees) LaTeX mark-up.	`[-p <op_name>]`
		Displays only the abstract test classes of operation schema `op_name`.
		`[-u <unfold_order>]`
		Same as before.
		`[-o <file_name>]`
		Same as before.

Command	Description	Options
`showspec`	Displays the entire specification LaTeX mark-up.	`[-u]` Same as `-u -1` before. `[-o <file_name>]` Same as before.
`showtt`	Displays all of the testing trees.	`[-p <op_name>]` Displays only the testing tree of operation schema `op_name`. `[-o <file_name>]` Same as before. `[-x]` Displays also the test classes that were pruned.

7.12 How to Quit Fastest

Important!! Before leaving Fastest save your results. Fastest will not save anything by default and will not remember you that there is unsaved data.

To leave the program just type `quit` and press the return key.

A Z Features Unsupported by Fastest

The following Z features are still unsupported by Fastest; the list is not exhaustive.

- The hide (`\hide`, $\backslash$) operator.

  Fastest will crash if the specification being loaded uses this operator.

- Schema names referenced in the predicate part of some schema.

  The referenced schemas will not be unfolded, thus severely reducing the effectiveness of both automatic pruning and abstract test case search. Test case design will still be quite meaningful.

- Variable substitution.

  Schema expressions such as $A[a/b]$ where A is a schema and a and b are variables, are not supported. If $B == A[a/b]$, then B will not be recognized as an operation regardless of A.

- The following operators: **if** , *min* and *max*.

  The **if** clause will not be rewritten when DNF is calculated, as it should be. If some expression of some satisfiable test class contains *min* or *max*, then it will be impossible to find an abstract test case for that test class.

- The LaTeX mark-up `\input`.

  Command `loadspec` will only load the specification explicitly present in the file passed as parameter. It will ignore any `\input` commands present in that file.

- Inductive types.

  Fastest is unable to find abstract test cases from test classes whose predicates include references to (non-constant) constructors defined in inductive types. Automatic pruning might not work correctly in this case.

  Though, enumerated types are fully supported.

- Schema types.

  All of the concepts related to schema types are unsupported, including the θ operator. Specifications containing schema types will be loaded and test case design will work mostly, but automatic pruning and abstract test case search will not work as usual.

  Section 6.8 might give some light on how to deal with this limitation.

- The Z sectioning system.

  Z sections are not recognized by Fastest.

- Generic definitions and generic schemas.

  This features are not supported although the user can perform test case design with specifications using them. Fastest will work as usual for those operation schemas that do not use generics.

- Schema composition and piping.

  Schemas defined by these operators will not be recognized as operations by Fastest, thus making it impossible for the user to "test" them.

  Section 6.4 might give some light on how to deal with this limitation.

B Test Classes Generated for *KeepMaxReading*

The following schema boxes represent the test classes generated for the operation *KeepMaxReading*. In the framework developed in [Sto93] each test class is described as a Z schema. This is important because only one notation is necessary to describe the specification and the test results.

KeepMaxReading_VIS

$$smax : SENSOR \pfun \mathbb{Z}$$
$$s? : SENSOR$$
$$r? : \mathbb{Z}$$

KeepMaxReading_DNF_1

$$smax : SENSOR \pfun \mathbb{Z}$$
$$s? : SENSOR$$
$$r? : \mathbb{Z}$$

$$s? \in \mathrm{dom}\ smax$$
$$smax\ s? < r?$$

KeepMaxReading_SP_1

$$smax : SENSOR \pfun \mathbb{Z}$$
$$s? : SENSOR$$
$$r? : \mathbb{Z}$$

$$s? \in \mathrm{dom}\ smax$$
$$smax\ s? < r?$$
$$smax\ s? < 0$$
$$r? < 0$$

KeepMaxReading_SP_2

$$smax : SENSOR \pfun \mathbb{Z}$$
$$s? : SENSOR$$
$$r? : \mathbb{Z}$$

$$s? \in \mathrm{dom}\ smax$$
$$smax\ s? < r?$$
$$smax\ s? < 0$$
$$r? = 0$$

KeepMaxReading_SP_3

$$smax : SENSOR \pfun \mathbb{Z}$$
$$s? : SENSOR$$
$$r? : \mathbb{Z}$$

$$s? \in \mathrm{dom}\ smax$$
$$smax\ s? < r?$$
$$smax\ s? < 0$$
$$r? > 0$$

KeepMaxReading_SP_4

$$smax : SENSOR \pfun \mathbb{Z}$$
$$s? : SENSOR$$
$$r? : \mathbb{Z}$$

$$s? \in \mathrm{dom}\ smax$$
$$smax\ s? < r?$$
$$smax\ s? = 0$$
$$r? > 0$$

KeepMaxReading_SP_5

$$smax : SENSOR \pfun \mathbb{Z}$$
$$s? : SENSOR$$
$$r? : \mathbb{Z}$$

$$s? \in \mathrm{dom}\ smax$$
$$smax\ s? < r?$$
$$smax\ s? > 0$$
$$r? > 0$$

KeepMaxReading_DNF_2

$$smax : SENSOR \pfun \mathbb{Z}$$
$$s? : SENSOR$$
$$r? : \mathbb{Z}$$

$$s? \notin \mathrm{dom}\ smax$$

KeepMaxReading_SP_6

$$smax : SENSOR \pfun \mathbb{Z}$$
$$s? : SENSOR$$
$$r? : \mathbb{Z}$$

$$s? \notin \mathrm{dom}\ smax$$
$$smax\ s? < 0$$
$$r? < 0$$

KeepMaxReading_SP_7

$$smax : SENSOR \pfun \mathbb{Z}$$
$$s? : SENSOR$$
$$r? : \mathbb{Z}$$

$$s? \notin \mathrm{dom}\ smax$$
$$smax\ s? < 0$$
$$r? = 0$$

KeepMaxReading_SP_8

$smax : SENSOR \pfun \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$

$s? \notin \operatorname{dom} smax$
$smax\ s? < 0$
$r? > 0$

KeepMaxReading_SP_9

$smax : SENSOR \pfun \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$

$s? \notin \operatorname{dom} smax$
$smax\ s? = 0$
$r? > 0$

KeepMaxReading_SP_10

$smax : SENSOR \pfun \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$

$s? \notin \operatorname{dom} smax$
$smax\ s? > 0$
$r? > 0$

KeepMaxReading_DNF_3

$smax : SENSOR \pfun \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$

$s? \in \operatorname{dom} smax$
$r? \leq smax\ s?$

KeepMaxReading_SP_11

$smax : SENSOR \pfun \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$

$s? \in \operatorname{dom} smax$
$r? \leq smax\ s?$
$smax\ s? < 0$
$r? < 0$

KeepMaxReading_SP_12

$smax : SENSOR \pfun \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$

$s? \in \operatorname{dom} smax$
$r? \leq smax\ s?$
$smax\ s? < 0$
$r? = 0$

KeepMaxReading_SP_13

$smax : SENSOR \pfun \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$

$s? \in \operatorname{dom} smax$
$r? \leq smax\ s?$
$smax\ s? < 0$
$r? > 0$

KeepMaxReading_SP_14

$smax : SENSOR \pfun \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$

$s? \in \operatorname{dom} smax$
$r? \leq smax\ s?$
$smax\ s? = 0$
$r? > 0$

KeepMaxReading_SP_15

$smax : SENSOR \pfun \mathbb{Z}$
$s? : SENSOR$
$r? : \mathbb{Z}$

$s? \in \operatorname{dom} smax$
$r? \leq smax\ s?$
$smax\ s? > 0$
$r? > 0$

C Standard Partitions

The following standard partitions are included by default in the file `lib/conf/stdpartition.spf`. The user can erase or modify these partitions and define new ones as well by simply editing the file. Fastest needs to be restarted so it is notified of changes.

C.1 Sets

Standard partition for expressions of the form $S \cup T$

$S = \{\}, T = \{\}$
$S = \{\}, T \neq \{\}$
$S \neq \{\}, T = \{\}$
$S \neq \{\}, T \neq \{\}, S \cap T = \{\}$
$S \neq \{\}, T \neq \{\}, S \subset T$
$S \neq \{\}, T \neq \{\}, T \subset S$
$S \neq \{\}, T \neq \{\}, T = S$
$S \neq \{\}, T \neq \{\}, (S \cap T) \neq \{\}, \neg (S \subseteq T), \neg (T \subseteq S), T \neq S$

Standard partition for expressions of the form $S \cap T$

$S = \{\}, T = \{\}$
$S = \{\}, T \neq \{\}$
$S \neq \{\}, T = \{\}$
$S \neq \{\}, T \neq \{\}, S \cap T = \{\}$
$S \neq \{\}, T \neq \{\}, S \subset T$
$S \neq \{\}, T \neq \{\}, T \subset S$
$S \neq \{\}, T \neq \{\}, T = S$
$S \neq \{\}, T \neq \{\}, (S \cap T) \neq \{\}, \neg (S \subseteq T), \neg (T \subseteq S), T \neq S$

Standard partition for expressions of the form $S \setminus T$

$S = \{\}, T = \{\}$
$S = \{\}, T \neq \{\}$
$S \neq \{\}, T = \{\}$
$S \neq \{\}, T \neq \{\}, S \cap T = \{\}$
$S \neq \{\}, T \neq \{\}, S \subset T$
$S \neq \{\}, T \neq \{\}, T \subset S$
$S \neq \{\}, T \neq \{\}, T = S$
$S \neq \{\}, T \neq \{\}, (S \cap T) \neq \{\}, \neg (S \subseteq T), \neg (T \subseteq S), T \neq S$

Standard partition for expressions of the form $x \notin A$

$A = \{\}$
$A \neq \{\}$

Standard partition for expressions of the form $x \in A$

$A = \{x\}$
$A \neq \{x\}, x \in A$

Standard partition for expressions of the form $\#A$

$\#A = 0$
$\#A = 1$
$\#A > 1$

C.2 Integers

Standard partition for expressions of the form $n < m$

$A < 0, B < 0$
$A < 0, B = 0$
$A < 0, B > 0$
$A = 0, B > 0$
$A > 0, B > 0$

Standard partition for expressions of the form $n \leq m$

$A < 0, B < 0, A < B$
$A < 0, B < 0, A = B$
$A < 0, B = 0$
$A < 0, B > 0$
$A = 0, B = 0$
$A = 0, B > 0$
$A > 0, B > 0, A < B$
$A > 0, B > 0, A = B$

Standard partition for expressions of the form $n > m$

$A < 0, B < 0$
$A < 0, B = 0$
$A = 0, B < 0$
$A > 0, B = 0$
$A > 0, B > 0$

Standard partition for expressions of the form $n \geq m$

$A < 0, B < 0, A > B$
$A < 0, B < 0, A = B$
$A = 0, B < 0$
$A = 0, B = 0$
$A > 0, B < 0$
$A > 0, B = 0$
$A > 0, B > 0, A > B$
$A > 0, B > 0, A = B$

Standard partition for expressions of the form $n = m$

$A < 0, B < 0$
$A = 0, B = 0$
$A > 0, B > 0$

Standard partition for expressions of the form $n \neq m$

$n < 0, m < 0$
$n < 0, m = 0$
$n < 0, m > 0$
$n = 0, m < 0$
$n = 0, m > 0$
$n > 0, m < 0$
$n > 0, m = 0$
$n > 0, m > 0$

Standard partition for expressions of the form $n + m$

$n < 0, m < 0, n < m$
$n < 0, m < 0, n = m$
$n < 0, m < 0, n > m$
$n < 0, m = 0$
$n < 0, m > 0$
$n = 0, m < 0$
$n = 0, m = 0$
$n = 0, m > 0$
$n > 0, m > 0, n < m$
$n > 0, m > 0, n = m$
$n > 0, m > 0, n > m$

C.3 Relations

Standard partition for expressions of the form $R \oplus G$

$R = \{\}, G = \{\}$
$R = \{\}, G \neq \{\}$
$R \neq \{\}, G = \{\}$
$R \neq \{\}, G \neq \{\}, \text{dom } R = \text{dom } G$
$R \neq \{\}, G \neq \{\}, \text{dom } G \subset \text{dom } R$
$R \neq \{\}, G \neq \{\}, (\text{dom } R \cap \text{dom } G) = \{\}$
$R \neq \{\}, G \neq \{\}, \text{dom } R \subset \text{dom } G$
$R \neq \{\}, G \neq \{\}, (\text{dom } R \cap \text{dom } G) \neq \{\}, \neg\, (\text{dom } G \subseteq \text{dom } R), \neg\, (\text{dom } R \subseteq \text{dom } G)$

Standard partition for expressions of the form $S \lhd R$

$R = \{\}$
$R \neq \{\}, S = \{\}$
$R \neq \{\}, S = \text{dom } R$
$R \neq \{\}, S \neq \{\}, S \subset \text{dom } R$
$R \neq \{\}, S \neq \{\}, S \cap \text{dom } R = \{\}$
$R \neq \{\}, S \cap \text{dom } R \neq \{\}, \text{dom } R \subset S$
$R \neq \{\}, S \cap \text{dom } R \neq \{\}, \neg\, (\text{dom } R \subseteq S), \neg\, (S \subseteq \text{dom } R)$

Standard partition for expressions of the form $S \lhd\!\!\!\!- R$

$R = \{\}$
$R \neq \{\}, S = \{\}$
$R \neq \{\}, S = \text{dom } R$
$R \neq \{\}, S \neq \{\}, S \subset \text{dom } R$
$R \neq \{\}, S \neq \{\}, S \cap \text{dom } R = \{\}$
$R \neq \{\}, S \cap \text{dom } R \neq \{\}, \text{dom } R \subset S$
$R \neq \{\}, S \cap \text{dom } R \neq \{\}, \neg\, (\text{dom } R \subseteq S), \neg\, (S \subseteq \text{dom } R)$

Standard partition for expressions of the form $R \rhd S$

$R = \{\}$
$R \neq \{\}, S = \{\}$
$R \neq \{\}, S = \mathrm{ran}\, R$
$R \neq \{\}, S \neq \{\}, S \subset \mathrm{ran}\, R$
$R \neq \{\}, S \neq \{\}, S \cap \mathrm{ran}\, R = \{\}$
$R \neq \{\}, S \cap \mathrm{ran}\, R \neq \{\}, \mathrm{ran}\, R \subset S$
$R \neq \{\}, S \cap \mathrm{ran}\, R \neq \{\}, \neg\, (\mathrm{ran}\, R \subseteq S), \neg\, (S \subseteq \mathrm{ran}\, R)$

D Elimination Theorems

This appendix lists the elimination theorems delivered with this version. All the elimination theorems except ExtensionalUndefinition and UndefinitionByEmptiness were certified with the Z/EVES proof assistant [Saa97]. Slightly weaker versions of the following elimination theorems were certified with Z/EVES: SingletonNotSet, SingletonIsNotEmpty, SingletonNotSubset, SingletonMappletNotInDom, SingletonNotSubsetDom, NrresCap, SingletonMappletNotEqualRel1, SingletonNotSubsetRel, ExcludedMiddleSingletonSub1 and NotinSetExtension.

Elimination Theorem NatDef $[n : \mathbb{N}]$

$\neg\ 0 \leq n$

Elimination Theorem Reflexivity $[x, \text{const } y : X]$

$x \neq y$
$x = y$

Elimination Theorem ExtensionalUndefinition $[f : X \pfun Y;\ x : X]$

$x \notin \text{dom}\, f$
somewhere$(f\ x)$

Elimination Theorem ArithmIneq1 $[\text{const } N : \mathbb{N};\ n, m : \mathbb{Z}]$

$n \leq m$
$m < N$
$n = N$

Elimination Theorem ArithmIneq2 $[\text{const } N : \mathbb{N};\ n, m : \mathbb{Z}]$

$n \leq m$
$m < N$
$n > N$

Elimination Theorem ArithmIneq3 $[\text{const } N : \mathbb{N};\ n, m : \mathbb{Z}]$

$n \leq m$
$m = N$
$n > N$

Elimination Theorem SingletonNotSet $[A : \mathbb{P}\, X;\ x : X]$

$x \notin A$
setExtension$(x) = A$

Elimination Theorem BasicMembershipContradiction $[A : \mathbb{P}\, X;\ x : X]$

$x \in A$
$x \notin A$

Elimination Theorem NotInEmptySet $[A : \mathbb{P}\, X;\ x : X]$

$x \in A$
$A = \{\}$

Elimination Theorem SingletonIsNotEmpty $[x : X]$

$\text{setExtension}(x) = \{\}$

Elimination Theorem NotSubsetOfSingleton $[A : \mathbb{P}\, X;\ x : X]$

$A \neq \{\}$
$A \subset \{x\}$

Elimination Theorem NotSubsetOfSingletonMapplet $[R : X \leftrightarrow Y;\ x : X;\ y : Y]$

$R \neq \{\}$
$\text{dom}\, R \subset \text{dom}\{x \mapsto y\}$

Elimination Theorem SingletonNotSubset $[A : \mathbb{P}\, X;\ x : X]$

$x \notin A$
$\text{setExtension}(x) \subset A$

Elimination Theorem DomNotSubsetOfSingleton $[R : X \leftrightarrow Y;\ x : X]$

$R \neq \{\}$
$\text{dom}\, R \subset \{x\}$

Elimination Theorem NotInEmptyDom $[R : X \leftrightarrow Y;\ x : X]$

$x \in \text{dom}\, R$
$R = \{\}$

Elimination Theorem BasicOrderingProperty $[n, m : \mathbb{Z}]$

$n \neq m$
$\neg\ n < m$
$\neg\ n > m$

Elimination Theorem UndefinitionByEmptiness $[f : X \nrightarrow Y]$

$f = \{\}$
somewhere(f anything)

Elimination Theorem SingletonMappletNotInDom $[R : X \leftrightarrow Y;\ x : X;\ y : Y]$

$x \notin \text{dom}\, R$
$\text{dom}\, \text{setExtension}(x \mapsto y) = \text{dom}\, R$

Elimination Theorem SingletonNotSubsetDom $[R : X \leftrightarrow Y;\ x : X;\ y : Y]$

$x \notin \text{dom}\, R$
$\text{dom}\, \text{setExtension}(x \mapsto y) \subset \text{dom}\, R$

Elimination Theorem NrresEmptyRel $[R : X \leftrightarrow Y;\ A : \mathbb{P}\, Y;\ x : X]$

$x \in \text{dom}(R \rhd A)$
$R = \{\}$

Elimination Theorem NrresCap $[R : X \leftrightarrow Y;\ A : \mathbb{P}\, Y;\ x : X]$

$x \in \mathrm{dom}(R \rhd A)$
$\mathrm{setExtension}(x) \cap \mathrm{dom}\, R = \{\}$

Elimination Theorem CardDomEmptyRel $[R : X \leftrightarrow Y;\ \mathrm{const}\ N, n : \mathbb{N}]$

$\mathrm{eval}(N > 0)$
$R = \{\}$
$n = N$
$\#\,\mathrm{dom}\, R = n$

Elimination Theorem CardRelSingleton $[R : X \leftrightarrow Y;\ \mathrm{const}\ N, n : \mathbb{N};\ r : X \times Y]$

$\mathrm{eval}(N > 1)$
$n = N$
$\#\,\mathrm{dom}\, R = n$
$\{r\} = R$

Elimination Theorem SingletonMappletNotEqualRel1 $[R : X \leftrightarrow Y;\ x : X;\ y : Y]$

$x \notin \mathrm{dom}\, R$
$\mathrm{setExtension}(x \mapsto y) = R$

Elimination Theorem SingletonNotSubsetRel $[R : X \leftrightarrow Y;\ x : X;\ y : Y]$

$x \notin \mathrm{dom}\, R$
$\mathrm{setExtension}(x \mapsto y) \subset R$

Elimination Theorem NotInEmptyRan $[R : X \leftrightarrow Y;\ y : Y]$

$y \in \mathrm{ran}\, R$
$R = \{\}$

Elimination Theorem SingletonMappletNotEqualRel2 $[R : X \leftrightarrow Y;\ x : X;\ \mathrm{const}\ y1, \mathrm{const}\ y2 : Y]$

$y1 \in \mathrm{ran}\, R$
$\{x \mapsto y2\} = R$

Elimination Theorem BasicSetContradiction $[A : \mathbb{P}\, X]$

$A = \{\}$
$A \neq \{\}$

Elimination Theorem ExcludedMiddleSingleton $[A : \mathbb{P}\, X;\ \mathrm{const}\ x, \mathrm{const}\ y : X]$

$\{x\} = A$
$\{y\} = A$

Elimination Theorem ExcludedMiddleSingletonSub1 $[A : \mathbb{P}\, X;\ \mathrm{const}\ x, \mathrm{const}\ y : X]$

$\{x\} = A$
$\mathrm{setExtension}(y) \subset A$

Elimination Theorem ExcludedMiddleSingletonSub2 $[A : \mathbb{P}\, X;\ \text{const}\ x, \text{const}\ y : X]$

$\{x\} = A$
$A \subset \{y\}$

Elimination Theorem RanNotSubsetOfSingleton $[R : X \leftrightarrow Y;\ y : Y]$

$R \neq \{\}$
$\text{ran}\, R \subset \{y\}$

Elimination Theorem SetComprNotEmpty1 $[\text{const}\ N : \mathbb{Z};\ A : \mathbb{P}\, X]$

$\text{eval}(N < 2)A \neq \{\}$
$\{\text{anything} : N \mathinner{..} \#A \bullet \text{anything}\} = \{\}$

Elimination Theorem SetComprNotASeq4 $[s : \text{seq}\, X;\ n : \mathbb{N}]$

$n = 0$
$s \neq \{\}$
$\text{dom}\, s = \text{dom}\{i : 1 \mathinner{..} \text{anything} \bullet i + n - 1 \mapsto \text{anything}\}$

Elimination Theorem SetComprNotASeq5 $[s : \text{seq}\, X;\ n : \mathbb{N}]$

$n = 0$
$s \neq \{\}$
$\text{dom}\{i : 1 \mathinner{..} \text{anything} \bullet i + n - 1 \mapsto \text{anything}\} \subset \text{dom}\, s$

Elimination Theorem NatRangeNotEmpty $[n, \text{const}\ N, \text{const}\ M : \mathbb{N}]$

$\text{eval}(N \leq M)$
$n + N \mathinner{..} (n + M) = \{\}$

Elimination Theorem ExcludedMiddle $[x, \text{const}\ y, \text{const}\ z : X]$

$x = y$
$x = z$

Elimination Theorem NotinSetExtension $[x, \text{const}\ y : X]$

$x \notin \text{setExtension}(y)$
$x = y$

Elimination Theorem SetComprIsEmpty1 $[R : X \leftrightarrow Y]$

$R = \{\}$
$\{\text{anything} : \text{dom}\, R \bullet \text{anything}\} \neq \{\}$

Elimination Theorem CapSubsetEmpty $[A, B : \mathbb{P}\, X]$

$A \neq \{\}$
$A \cap B = \{\}$
$A \subset B$

Elimination Theorem CapEqEmpty $[A, B : \mathbb{P}\, X]$

$A \neq \{\}$
$A \cap B = \{\}$
$A = B$

Elimination Theorem CapEmpty $[A, B : \mathbb{P}\, X]$
$A = \{\}$
$A \cap B \neq \{\}$

Elimination Theorem NatRangeNotEmpty2 $[\text{const } N, \text{const } M : \mathbb{N}]$
$\text{eval}(N < M)$
$N \mathinner{..} M = \{\}$

Elimination Theorem NatRangeNotEmpty3 $[n, m, \text{const } N, \text{const } M : \mathbb{N}]$
$n + (m * N) \mathinner{..} (n + ((m + M) * N)) = \{\}$

Elimination Theorem NatRangeNotSubset $[n, m, \text{const } N, \text{const } M, \text{const } P, \text{const } Q : \mathbb{N}]$
$\text{eval}(M - N > Q * P)$
$N \mathinner{..} M \subset n + (m * P) \mathinner{..} (n + ((m + Q) * P))$

Elimination Theorem NatRangeNotEq $[n, m, \text{const } N, \text{const } M, \text{const } P, \text{const } Q : \mathbb{N}]$
$\text{eval}(M - N > Q * P)$
$N \mathinner{..} M = n + (m * P) \mathinner{..} (n + ((m + Q) * P))$

Elimination Theorem NatRangeNotEmpty5 $[n, m, \text{const } P, \text{const } Q : \mathbb{N}]$
$\text{eval}(Q * P > 0)$
$n + (m * P) \mathinner{..} (n + ((m + Q) * P)) = \{\}$

Elimination Theorem NatRangeNotEmpty4 $[n, m : \mathbb{N}]$
$n \mathinner{..} (n + m) = \{\}$

Elimination Theorem NatRangeNotSubset2 $[n, \text{const } N, \text{const } M, \text{const } P : \mathbb{N}]$
$\text{eval}(M - N > P)$
$N \mathinner{..} M \subset (n \mathinner{..} (n + P))$

Elimination Theorem NatRangeNotEq2 $[n, \text{const } N, \text{const } M, \text{const } P : \mathbb{N}]$
$\text{eval}(M - N > P)$
$N \mathinner{..} M = (n \mathinner{..} (n + P))$

Elimination Theorem NatRangeNotSubset3 $[n, m, p, \text{const } N, \text{const } M, \text{const } P, \text{const } Q : \mathbb{N}]$
$\text{eval}(M - N > Q * P)$
$m \leq (n + Q) * P$
$N \mathinner{..} M \subset p + (n * P) \mathinner{..} (p + m)$

Elimination Theorem NatRangeNotEq3 $[n, m, p, \text{const } N, \text{const } M, \text{const } P, \text{const } Q : \mathbb{N}]$
$\text{eval}(M - N > Q * P)$
$m \leq (n + Q) * P$
$N \mathinner{..} M = p + (n * P) \mathinner{..} (p + m)$

References

[Abr96] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[CAR10] Maximiliano Cristiá, Pablo Albertengo, and Pablo Rodríguez Monetti. Ipruning testing trees in the test template framework by detecting mathematical contradictions. In *SEFM*, 2010.

[CR09] Maximiliano Cristiá and Pablo Rodríguez Monetti. Implementing and applying the Stocks-Carrington framework for model-based testing. In Karin Breitman and Ana Cavalcanti, editors, *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 167–185. Springer, 2009.

[Dil90] Antoni Diller. *Z: An Introduction to Formal Methods*. John Wiley Press, 1990.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of sofware engineering*. Prentice Hall, Upper Saddle River, New Jersey, 1991.

[HP95] Hans Martin Hörcher and Jan Peleska. Using Formal Specifications to Support Software Testing. *Software Quality Journal*, 4:309–327, 1995.

[ISO02] ISO. Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics. Technical Report ISO/IEC 13568, International Organization for Standardization, 2002.

[Jac97] Jonathan Jacky. *The Way of Z*. Cambridge University Press, 1997.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2002.

[PST96] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. Prentice Hall International, 1996.

[Saa97] Mark Saaltink. The Z/EVES System. In J.P. Bowen, M.G. Hinchey, and D. Till, editors, *ZUM '97: The Z Formal Specification Notation*, pages 72–85, 1997.

[SC96] P. Stocks and D. Carrington. A Framework for Specification-Based Testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[Sto93] P. Stocks. *Applying Formal Methods to Software Testing*. PhD thesis, Department of Computer Science, University of Queensland, 1993.

[UL06] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[WD96] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.